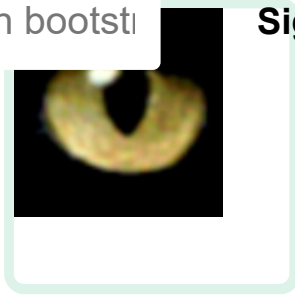




Search bootst

Sign up

Follow bootstrap-zero



Bootstrap Zero

POSTS

ARCHIVE

Hi there

You've reached the Bootstrap Zero blog, which was written during July 2015 for [RetroChallenge 2015/07](#).

If you keep reading down this page, you'll be reading the posts in reverse order, and they might not make much sense that way.

It'd probably be better to read [the archived version of this blog](#) in [the SITU-SOL repository](#), where you will also find continuation of this work, now and in future.

Thank you for your interest in writing code by hand!

1 note Oct 16th, 2017

MORE YOU MIGHT LIKE

Retrospective

What have I learned from this? What would I do differently, if I had to do it all again?

Writer's cramp. I really didn't foresee that. I probably should've, but I didn't. My wrist seems OK now, but last week it was definitely a bit ache-y, and I scaled back my ambitions a bit because I didn't want to stress it too much. I guess it's been a while since I've written this much.

Uneven columns of text. I wrote the "nice" version of the code on printer paper placed over lined paper, to keep my rows straight, but I really should've used graph paper instead, to keep my columns straight too. Or French ruled paper, or a ledger, or something. Also, putting more thought into choice of colours, etc., would've been nice.

Assembler-ness in hand-assembly. I only flaunted the fact that I didn't have to please an assembler's idea of syntax a little bit: I used the same "local labels" like NEXT: over and over again, since I knew very well which one I meant at each point. An assembler program would've wanted me to call them NEXT1, NEXT2 or whatever, to keep them apart. I even used underscores in these labels, purely out of habit, but of course they weren't at all necessary. I could've even gone much further – I could've just drawn arrows from one part of the code to another. I kind of regret not doing something more "artistic" like that.

No LEDs for the address lines. As I mentioned in the I love bits post, it would've saved a significant amount of debugging if I had known *what* address I was entering bits *at*. That's a definite advantage the nice big front-panel of the Altair 8800 has over the austere front-panel of the 1802 Membership Card. But even showing

the lowest 8 bits of the address would've been useful as a guard against this sort of address-misalignment problem.

No padding between routines. I hand-assembled each routine to start at the very next address available after the previous routine. My excuse was that I didn't want to have to enter more bytes than I had to. But this is a risky practice, because it's quite easy to miss an instruction, or to want to insert an instruction later, for debugging; but if there's no free space after the routine, you need to figure out what to replace with a JMP to somewhere else in memory (which will JMP back to continue the routine), then you need to find somewhere else in memory to put it... it's a mess. I was lucky in that I didn't hit a problem that required this. But it was luck. It would've been smarter to add 3 or so NOPs after the end of each routine, to make adjusting the routine easier, if needed. Even if it would've meant entering 24 more bits each time.

No comments. Yeah, that's the thing, see. Writing documentation by hand is harder than writing code by hand because documentation is... sentences. The fact that you're already writing out the "important" stuff seems like a good excuse to leave out the "non-critical" stuff like comments. But the comments *are* important – especially in assembly language, where it's really important to define what each routine expects when it is called, and what the rest of the program can expect when it's finished (preconditions and postconditions.) If you fail to pay attention to these when writing code that calls that routine, very bad things can happen. Most of my first drafts did include scribbled preconditions and postconditions, but those notes didn't make it to the final listings.

The language is a bit goofy. And incomplete (no real standard library.) Well, I kept going back and forth on whether I should make a FORTH-like language or a LISP-like language, while at the same

time wanting to keep it as simple as possible... and eventually simplicity won out. The resulting design for SITU-SOL has some similarities to Shelta, which I suppose isn't surprising.

...

OK, those are the basic ones. But the big one is:

A programming language alone does not a programming environment make!

See, one reasonable way to interpret the goal of this project of writing code by hand, is to write enough code by hand so that you don't have to write code by hand anymore.

But SITU-SOL doesn't accomplish that goal. If you wanted to write a non-trivial SITU-SOL program... you'd probably want to write it out by hand first!

:(

In other words... I probably should've just written a text editor instead. Maybe next year?

Well, that's not quite a fair assessment, either. Let me explain...

In the modern world, we'd write our program in our text editor, and we'd assemble it with our assembler program, which is a separate thing.

But uncompressed text takes up a fair amount of space, and on 8-bit machines, with their modest RAM and slow (or non-existent) persistent storage, that space is at a premium. So programming environments like BASIC do clever tricks like **tokenizing** the source code; an instruction like INPUT, even though it is entered as five characters, can be represented by one byte in memory, and

displayed as those five characters again when the program is LISTed. Forth can do a similar trick (although I admit I'm not familiar with exactly how Forth is spec'ed to work, in this regard.)

So, once I got SITU-MON in, I probably should've written a text editor, so that I could design and write and edit further programs directly on the computer, without using paper.

But once I came across the idea that SITU-SOL's code could be *both* compiled *and* tokenized (i.e., treat JSR \$xxxx as a "token" that prints out the symbol-name just before \$xxxx when it is decompiled, and so forth) – that's pretty neat, and I felt I had to go in that direction. (Maybe that's how Forth works, too. I should look that up.)

Anyway...

To achieve this "edit directly on the computer, avoid paper" plateau, there are at least 3 more things the SITU-SOL environment should do:

- Let you list the symbols that have been defined so far. (Like Forth's WORDS, but I guess it would be called \$SYMBOLS here.)
- Let you see the contents of a symbol's definition, in a human-readable format.
- Let you change the definition of a symbol. Or, if you can't do that directly, let you replace every call to one symbol, with a call to a different symbol, in every defined symbol. (Then instead of changing a symbol, you write a new symbol and replace all the calls to the old one.)

Now, I *did* design SITU-SOL to support these things... and I *have* written code to do these things! I include it below, for completeness. But I don't think I'll have the time (or the wrist-stamina) to hand-

assemble it and enter it before the end of the month.

The code for \$SYMBOLS:

```
SYMBOLS:
WORDS:      JSR  RESET_STAB           ← define a
                JSR  RESET_LINE       symbol
                LOOP: JSR  PEEK_STAB   $SYMBOLS
                BNE  CONT              which JSR's this
                JSR  OUTPUTLN_TTY
                CONT: JSR  WRITE_STAB_NAME
                JMP  LOOP
```

```
WRITE_STAB_NAME: JSR  PEEK_STAB
                  AND  #$80
                  BNE  SET
                  JSR  WRITE_CHAR
                  JSR  INCR_STAB_PTR
                  JMP  WRITE_STAB_NAME
                SET: JSR  INCR
                  JSR  PEEK
                  JSR  ADD_A_STAB
                  JSR  INCR
                  RTS
```

Dumping the contents of a symbol:

```
WRITE_SPACE:  LDA #$24
               JSR WRITE_CHAR
```

assumes STAB_PTR is at beginning of payload of symbol to dump

```
DUMP_SYMBOL:  JSR READ_STAB_PTR
               CMP #$A9
               BNE NEXT
               LDA #'?'
               JSR WRITE_CHAR
               JSR INCR_STAB_PTR
               JSR READ_STAB
               JSR WRITE_BYTE
               JMP WRITE_SPACE
```

```
WRITE_SPACE:  LDA #$24
               JSR WRITE_CHAR
               JMP DUMP
```

```
NEXT:  CMP #$EA
        BNE NEXT
        LDA #'?'
        JSR WRITE_CHAR
        LDA #3
        JSR ADD_A_STAB
        JMP WRITE_SPACE
```

```
LDA #2
JSR ADD_A_STAB
```

~~expl of SYMBOL + STAB~~
~~stacked SYMBOLS, to get~~
~~impl notes: contents of symbol~~
~~doc of MATCH/LOOKUP~~
~~lessons: text ed/asm~~

~100
bytes

```
NEXT:  CMP #$D0
        BNE NEXT
        dup of $ w/ #'! LDA #'! ; JMP
```

(double-check this)

```
NEXT:  CMP #$00
        BNE NEXT
        JSR INCR
        JSR READ
        STA ARG_LO
        JSR INCR
        JSR READ
        STA ARG_HI
        JSR PRINT SYM_IN_ARG
        JSR INCR
        JMP WRITE_SPACE

NEXT:  CMP #$60
        BNE ERROR
        RTS

ERROR:  LDA #'@
        JMP WRITE_CHAR
```

```

PRWT-SYM-IN-ARG:  MACRO
JSR SAVE_STAB_PTR
MACRO JSR COPY_ARG_TO_STAB_PTR
LDA #2
STA STAB_DELTA
JSR SUB_STAB_PTR
JSR READ_STAB
AND #$7F
STA STAB_DELTA
JSR SUB_STAB_PTR
JSR READ_STAB
AND #$80
BNE END
JSR READ_STAB
JSR WRITE_CHAR
JSR INCR_STAB
JMP LOOP
END:  MACRO JSR RESTORE_STAB_PTR
RTS

~58
bytes

SAVE_STAB:          COPY_ARG_TO_STAB:
LDA STAB_PTR_LO      LDA ARG_LO
PHA                  STA STAB_PTR_LO
LDA " " _HI          LDA _HI
PHA                  STA _HI

RESTORE:
PLA
STA " " _HI
PLA
STA " " _LO

```

Replacing all calls to symbol1 with calls to symbol2:

GIVEN STAB_PTR POINTING TO THE ^{BEGINNING} ~~CONTENTS~~ OF A SYMBOL TO REWRITE,

ARG_PTR containing the SEARCH address, NEW_PTR containing the REPLACEMENT,

REPLACE_CALLS: JSR SKIP_TO_CONTENTS

~~REPLACE_CALLS~~ ^{LOOP}: JSR PEEK_STAB

CMP #\$00

~~BNE~~ CONT

RTS

← JSR INCR_STAB → postcondition: STAB_PTR
points to beginning of
next symbol

CONT: CMP #\$60

BEQ REPLACE_TRY

LDA #\$03

JSR ADD_PTR_STAB_PTR

JMP ~~REPLACE_CALLS~~ LOOP

REPLACE_TRY: JSR INCR_STAB

~~LDA NEW~~

JSR PEEK_STAB

CMP ARG_PTR_LO

BEQ ONE_MATCH

~~JMP~~ LDA #\$02

JMP

ONE_MATCH: JSR INCR_STAB

JSR PEEK_STAB

CMP ARG_PTR_HI

BEQ MATCH

LDA #\$01

JMP

MATCH: LDA #\$01

JSR SUB_STAB

LDA NEW_LO

JSR WRITE_STAB

(?) → LDA NEW_HI

JSR WRITE_STAB

JMP ~~REPLACE_CALLS~~
LOOP

SKIP_TO_CONTENTS: JSR PEEK_STAB

AND #\$80

BNE SET

JSR INCR_STAB

JMP SKIP_TO_CONTENTS

SET: LDA #\$02

JMP ADD_A_STAB_PTR

Same preconditions about ARG_ and NEW_ as REPLACE_CALLS

```
REPLACE_ALL_CALLS: JSR RESET_STAB_PTR
                   Loop: JSR PEEK_STAB
                        BNE CONT
                        RTS
                   CONT: JSR REPLACE_CALLS
                        JMP Loop
```

```
CMD_REPLACE_CALLS: JSR LOOKUP_SYMBAL
                   BCS FAIL
                   LDA STAB_PTR_LO
                   STA ARG_PTR_LO
                   LDA _HI
                   STA _HI
                   JSR LOOKUP_SYMBAL
                   BCS FAIL
                   LDA STAB_LO
                   STA NEW_LO
                   LDA _HI
                   STA _HI
                   JMP REPLACE_ALL_CALLS
                   FAIL: JMP REPORT_ERROR
```

Calling those last two routines from the interactive prompt, with /D SYMBAL to dump and /R SYMBAL1 SYMBAL2 to replace calls to symbols:

```

CMD_DUMP_SYMBOL:  JSR  LOOKUP_SYMBOL          12 bytes
                   JSR  RESET_LINE
                   JSR  DUMP_SYMBOL
                   JMP  OUTPUTLN_TTY

```

```

SLASH_CMDS:       (INCN LINE IF WE HAVEN'T FROM / )
                   JSR  READ_CHAR
                   CMP  #'D
                   BNE  NEXT
                   JMP  CMD_DUMP_SYMBOL

```

```

NEXT:             CMP  #'R
                   BNE  NEXT
                   JMP  CMD_REPLACE_CALLS

```

```

NEXT:             JMP  REPORT_ERROR    ← change to
                                         continuation, for
                                         expanded version

```

So, if the point of this project was to write the code by hand, I can indeed say I have written those routines by hand! Not assembled, entered, or tested... but written, yes!

Future work

I've created a [Mercurial repository on Bitbucket](#), which is mirrored in a [git repository on Github](#), where I'll be placing the support materials and binary images from this project, and maybe in the future... other stuff, like a disassembly of the code that was entered.

One of the reasons I entered SITU-MON at \$8100 was because when the Commodore 64 is booted with an 8K cartridge attached, that ROM appears at \$8000. I didn't start right at \$8000 because I wanted to reserve room for the header (the bit of ROM that tells the C64 "yes, this is a cartridge") and some routines to do some initialization.

So, one thing I'd really like to do is make such a binary image. Even if only to use it in the emulator. Because it'd be kind of cool to boot the machine directly into SITU-SOL. But that's a bit outside of the scope of this RetroChallenge project.

But after July ends.... who knows?

1 note

SITU-SOL

SITU-SOL

THE DIFFERENCE BETWEEN A "LOW-LEVEL" PROGRAMMING LANGUAGE AND A "HIGH-LEVEL" ONE IS NOT CLEAR-CUT, BUT ONE THING THAT MOST "HIGH-LEVEL" LANGUAGES DO IS TO ALLOW THE PROGRAMMER TO DEFINE THEIR OWN IDENTIFIERS USING A (MOSTLY) FREE CHOICE OF LETTERS AND NUMBERS. IMPLEMENTING THIS FEATURE MEANS STORING THESE SYMBOLS* IN A DATA STRUCTURE, AND RETRIEVING THEM FROM THIS STRUCTURE AS WELL. WE'LL IMPROVISE A SYMBAL-ORIENTED LANGUAGE AROUND THIS, AND CALL IT:

SITU-SOL.

* WHY IS IT SPELLED THIS WAY? WELL,

1. THE COMMODORE'S OPERATING SYSTEM IS CALLED THE "KERNAL", AND

2. ACCORDING TO THE INTRODUCTION OF THE BOOK INTERACTIVE FICTION AND ADVENTURE GAMES FOR MICROCOMPUTERS: AN ANNOTATED DIRECTORY, 1988, THE PART OF SUCH A PROGRAM WHICH HANDLES THE DIALOGUE OR INTERACTION WITH THE PLAYER IS CALLED THE "PARSAR".

BY PATRICK R. DEWEY
ISBN 0-88736-170-6

THEREFORE, WE'RE GOING TO HAVE A SYMBAL TABLE, OBVIOUSLY.

(NOTE, ALSO, THAT IN THE PROGRAM LISTINGS, "SYMBAL TABLE" IS FREQUENTLY ABBREVIATED AS "STAB". THIS IS PURELY FOR CONVENIENCE AND SHOULD NOT BE CONSTRUED AS AN ACT OF STRUCTURAL VIOLENCE.)

SYNTAX OF SITU-SOL

A SITU-SOL PROGRAM CONSISTS OF A NUMBER OF LINES, WHICH MAY BE ENTERED INTERACTIVELY. THE FIRST CHARACTER OF EACH LINE HAS A SPECIAL MEANING:

[symbol symbol ... symbol

- COMPILE THE SYMBOLS TO THE COMPILATION BUFFER.

] symbol

- DEFINE A NEW SYMBOL, USING THE CURRENT CONTENTS OF THE COMPIL. BUFFER.

/ ...

- RESERVED FOR FUTURE EXP.

.

- QUIT SITU-SOL (AND RETURN TO SITU-MON, IF APPLICABLE)

IF THE LINE BEGINS WITH ANY OTHER CHARACTER, IT IS NOT SPECIAL, IT IS SIMPLY

symbol symbol ... symbol

THESE SYMBOLS ARE COMPILED TO THE BUFFER AND IMMEDIATELY EXECUTED (IMMEDIATE MODE.)

COMPILATION OF SYMBOLS

SOME SYMBOLS ARE SPECIAL FORMS WITH SPECIAL MEANINGS:

#xy - READ xy AS A BYTE (TWO HEXADECIMAL DIGITS) AND LOAD THAT BYTE INTO THE ACCUMULATOR.

? symbol - EXECUTE THE NEXT SYMBOL ONLY IF THE ZERO FLAG IS NOT SET. ("IF")

symbol ! - EXECUTE THE PREVIOUS SYMBOL AGAIN (AND AGAIN...) IF THE ZERO FLAG IS NOT SET. ("REPEAT")

SYMBOLS BEGINNING WITH < OR > HAVE A SPECIAL MEANING WHICH IS RESERVED.

OTHERWISE, THE SYMBOL IS LOOKED UP IN THE SYMBOL TABLE, AND THE DEFINITION ASSOCIATED WITH THAT SYMBOL IS EXECUTED. IF THE SYMBOL COULD NOT BE FOUND IN THE SYMBOL TABLE, THAT'S A COMPILATION ERROR.

STANDARD SYMBOLS

WE WILL NEED A SET OF PRE-DEFINED SYMBOLS IN ORDER TO DO ANYTHING INTERESTING IN THIS LANGUAGE.

UNFORTUNATELY, I HAVEN'T REALLY WORKED OUT WHAT THAT SET WILL BE, YET.

WE'LL DEFINE SOME SIMPLE AD-HOC SYMBOLS FOR TESTING PURPOSES LATER ON...

FOR NOW, I'LL NOTE THAT WE COULD DEFINE A SET OF SYMBOLS THAT OPERATE ON A STACK, MAKING THE LANGUAGE VAGUELY FORTH-LIKE.

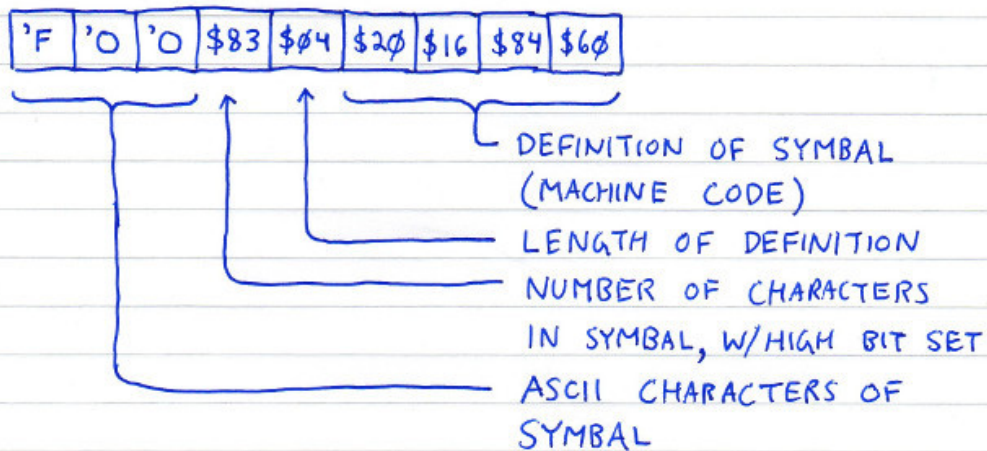
OR, IF WE DEFINE A TAPE-LIKE STORAGE FACILITY, AND HAVE SYMBOLS LIKE:

\$LEFT
\$RIGHT
\$CMP
\$NOT
\$WRITE

... THEN WE OUGHT TO BE ABLE TO CODE ANY TURING MACHINE IN SITU-SOL, MAKING THE LANGUAGE TURING-COMPLETE.

IMPLEMENTATION NOTES.

THE SYMBOL TABLE BEGINS AT \$0900 AND GROWS UPWARD.
EACH ENTRY LOOKS LIKE:



COMPILING A SYMBOL IS DONE BY WRITING A JSR
TO THE FIRST BYTE OF THE DEFINITION OF THE
SYMBOL. GIVEN ONLY THIS ADDRESS, THE NAME
CAN BE RECOVERED BY:

- SUBTRACT 2 FROM THIS ADDRESS → ADDRESS
- READ BYTE FROM THERE, "AND" IT WITH \$7F → LEN
- SUBTRACT LEN FROM THE ADDRESS → START
- READ BYTES FROM THERE UNTIL YOU SEE A HIGH BIT
(OR JUST READ "LEN" BYTES FROM THERE)

THERE ARE THE FOLLOWING FURTHER RESTRICTIONS ON THE MACHINE-CODE DEFINITION OF A SYMBOL:

- IT MUST CONSIST OF $3n+1$ BYTES, WHERE n IS AN INTEGER (NON-NEGATIVE, OF COURSE) AND THE LAST BYTE IS $\$G\emptyset$ (RTS).
- ANY SYMBOL WHOSE NAME BEGINS WITH THE CHARACTER '\$' WILL NOT BE DECOMPILED OR RELOCATED BY THE SYSTEM (THESE ARE PRE-DEFINED "SYSTEM SYMBOLS".)

THE FIRST RESTRICTION IS SO THAT IT'S EASY TO DECOMPILE A SYMBOL (I.E. TREAT THIS AS BOTH COMPILED AND TOKENIZED CODE.) WE COMPILE EACH SYMBOL OR SPECIAL FORM TO A MACHINE-CODE INSTRUCTION, AND PAD IT TO 3 BYTES WITH NOP'S. WHEN DECOMPILING, WE LOOK FOR 3-BYTE SEQUENCES THAT WE GENERATED AND TRANSLATE THEM TO TEXT.

THE SECOND RESTRICTION IS SO THAT WE CAN HAVE SYMBOLS THAT VIOLATE THE FIRST RESTRICTION; FOR EXAMPLE, CONTAINING JSR'S TO MACHINE-CODE ROUTINES ELSEWHERE IN MEMORY. BECAUSE THESE ROUTINES ARE NOT IN A SYMBOL TABLE, DECOMPILED WOULD PRODUCE GARBAGE WHEN IT TRIED TO FIND OUT THEIR NAMES AND PRINT THEM.

Now, the program listing! No, wait – I don't want you to get the impression that this just appeared fully-formed on paper. So, first, I'll ask you to imagine about 30 sheets of paper that looked something

like this:

MATCH_WORD: inputs: DICT_PTR and LINE-OFF — outputs: DICT_PTR, LINE-OFF, CARRY flag.
 If the token at LINE-OFF matches the word at DICT_PTR,
 return with CARRY clear and DICT_PTR ~~ENTRY-ENTRY~~
 pointing at the entry size byte of the dict entry.
 If the token does not match, return with CARRY set,
 advance DICT_PTR beyond the current entry (i.e. to the next entry),
 and leave LINE-OFF as it was. Assumes token followed by a space.
 or return.

```

LDX LINE-OFF
LDA SPACE #20
CMP LINE, X
BEQ PAST-TOKEN
LDA (DICT_PTR), Y
CMP LINE, X
BNE CHAR_MISMATCH
JSR INCR_DICT_PTR
INX
JMP

LDX LINE-OFF
LDA LINE, X
CMP #20
BEQ MATCH
CMP #00
BEQ MATCH
LDY #0
LDA (DICT_PTR), Y
PHA
AND #80
BNZ CHECK_EOT
PLA
CMP LINE, X
BNE CHAR_MISMATCH
JSR INCR_DICT_PTR
INX
JMP

LDA LINE, X
CMP #20
BEQ MATCH
CMP #00
BEQ MATCH
LDY #0
LDA (DICT_PTR), Y
AND #80
BNZ SKIP_ENTRY
JSR INCR_DICT_PTR
JMP
STX LINE-OFF
CLC
RTS

LDA (DICT_PTR), X
AND #7F
TAX
JSR ADD_X_TO_DICT_PTR
SEC
RTS
  
```

OK, now the program listing!

8300	RESET_STAB_PTR:	LDA #00	A9	00
2		STA STAB_PTR_LO	85	0A
4		LDA #09	A9	09
6		STA STAB_PTR_HI	85	0B
8		RTS	60	
8309	INCR_STAB_PTR:	LDA #01	A9	01
830B	ADD_A_STAB_PTR:	STA STAB_DELTA	85	0C
830D	ADD_TO_STAB_PTR:	CLC	18	
E		LDA STAB_PTR_LO	A5	0A
10		ADC STAB_DELTA	65	0C
2		STA STAB_PTR_LO	85	0A
4		LDA STAB_PTR_HI	A5	0B
6		ADC #00	69	00
8		STA STAB_PTR_HI	85	0B
A		RTS	60	
831B	SUB_STAB_PTR:	SEC	38	
C		LDA STAB_PTR_LO	A5	0A
E		SBC STAB_DELTA	E5	0C
20		STA STAB_PTR_LO	85	0A
2		LDA STAB_PTR_HI	A5	0B
4		SBC #00	E9	00
6		STA STAB_PTR_HI	85	0B
8328		RTS	60	

8329	PEEK_STAB:	LDY #\$00	A0	00	
B		LDA (STAB_PTR), Y	B1	0A	
D		STA STAB_BYTE	85	0D	
F		RTS	60		
8330	WRITE_STAB:	LDY #\$00	A0	00	
2		STA (STAB_PTR), Y	91	0A	
4		JMP INCR_STAB_PTR	4C	<u>09</u>	<u>83</u>
8337	LOOKUP_SYMBAL:	JSR RESET_STAB_PTR	20	<u>00</u>	<u>83</u>
A	LOOP:	JSR PEEK_STAB	20	<u>29</u>	<u>83</u>
D		BEQ END_OF_STAB	F0	<u>0C</u>	
F		JSR MATCH_SYMBAL	20	<u>4D</u>	<u>83</u>
42		BCS LOOP	B0	<u>F6</u>	
4		LDA #\$02	A9	02	
6		JSR ADD_ ~~A~~ -STAB_PTR	20	<u>0B</u>	<u>83</u>
9		CLC	18		
A		RTS	60		
B	END_OF_STAB:	SEC	38		
834C		RTS	60		

834D	MATCH_SYMBOL:	LDX LINE_OFF	A6 FE
834F	LOOP:	JSR PEEK_STAB	20 29 83
52		AND #\$80	29 80
4		BNE CHECK_END	D0 0E
6		LDA STAB_BYTE	A5 0D
8		CMP LINE, X	DD 00 C0
B		BNE MISMATCH	D0 12
D		JSR INCR_STAB_PTR	20 09 83
60		INX	E8
1		JMP LOOP	4C 4E 83
4	CHECK_END:	LDA LINE, X	BD 00 C0
7		CMP #\$20	C9 20
9		BEQ MATCH	F0 11
B		CMP #\$0D	C9 0D
D		BEQ MATCH	F0 0D
836F	MISMATCH:	JSR PEEK_STAB	20 29 83
72		AND #\$80	29 80
4		BNE SKIP_ENTRY	D0 0A
6		JSR INCR_STAB_PTR	20 09 83
9		JMP MISMATCH	4C 6F 83
C	MATCH:	STX LINE_OFF	86 FE
E		CLC	18
F		RTS	60
80	SKIP_ENTRY:	JSR INCR_STAB_PTR	20 09 83
3		JSR PEEK_STAB	20 29 83
6		STA STAB_DELTA	85 0C
8		INC STAB_DELTA	E6 0C
A		JSR ADD_TO_STAB_PTR	20 0D 83
D		SEC	38
838E		RTS	60

838F	PEEK_LINE:	LDY	LINE_OFF	A4	FE	
91		LDA	LINE, Y	~~A0~~	~~FE~~	
4		RTS		B9	00	C0
				60		

8395	RESET_LINE:	LDA	#\$00	A9	00	
7		STA	LINE_OFF	85	FE	
9		RTS		60		

839A	SKIP_SPACE:	JSR	PEEK_LINE	20	<u>8F</u>	<u>83</u>
D		CMP	#\$0D	C9	0D	
F		BEQ	CR	F0	<u>06</u>	
A1		CMP	#\$20	C9	20	
3		BEQ	SPACE	F0	<u>04</u>	
5		CLC		18		
6		RTS		60		
7	CR:	SEC		38		
8		RTS		60		
9	SPACE:	INC	LINE_OFF	E6	FE	
B		JMP	SKIP_SPACE	4C	<u>9A</u>	<u>83</u>

83AE	READ_ARG :	JSR	READ_BYTE	20	<u>4D</u>	<u>81</u>
B1		STA	ARG_HI	85	<u>0F</u>	
3		JSR	READ_BYTE	20	<u>4D</u>	<u>81</u>
6		STA	ARG_LO	85	<u>0E</u>	
83B8		RTS		60		

83B9	RESET_EMIT:	LDA	#\$00	A9	00	
B		STA	EMIT_OFF	85	10	
D		RTS		60		
83BE	EMIT_BYTE:	LDX	EMIT_OFF	A4	10	
C0		STA	EMITBUF, Y	99	00	C1
3		INC	EMIT_OFF	E6	10	
5		RTS		60		
83C6	EMIT_CALL:	LDA	#\$20	A9	20	
8		JSR	EMIT_BYTE	20	BE	83
B		LDA	STAB_PTR_LO	A5	0A	
D		JSR	EMIT_BYTE	20	BE	83
D0		LDA	STAB_PTR_HI	A5	0B	
2		JMP	EMIT_BYTE	4C	BE	83
83D5	EMIT_IF:	LDA	#\$EA	A9	EA	
7		JSR	EMIT_BYTE	20	BE	83
A		LDA	#\$F0	A9	F0	
C		JSR	EMIT_BYTE	20	BE	83
F		LDA	#\$03	A9	03	
E1		JMP	EMIT_BYTE	4C	BE	83
83E4	EMIT_REPEAT:	LDA	#\$D0	A9	D0	
6		JSR	EMIT_BYTE	20	BE	83
9		LDA	#\$FB	A9	FB	
B		JSR	EMIT_BYTE	20	BE	83
E		LDA	#\$EA	A9	EA	
83F0		JMP	EMIT_BYTE	4C	BE	83

83F3	EMIT_CONST:	LDA	#\$A9	A9	A9	
5		JSR	EMIT_BYTE	20	BE	83
8		LDA	ARG_LO	A5	0E	
A		JSR	EMIT_BYTE	20	BE	83
D		LDA	#\$EA	A9	EA	
F		JMP	EMIT_BYTE	4C	BE	83

8402	EMIT_LOAD:	LDA	#\$AD	A9	AD	
8404	EMIT_ARG:	JSR	EMIT_BYTE	20	BE	83
7		LDA	ARG_LO	A5	0E	
9		JSR	EMIT_BYTE	20	BE	83
C		LDA	ARG_HI	A5	0F	
E		JMP	EMIT_BYTE	4C	BE	83

8411	EMIT_STORE:	LDA	#\$8D	A9	8D	
3		JMP	EMIT_ARG	4C	<u>04</u>	<u>84</u>

8416	WRITE_PROMPT:	JSR	RESET_LINE	20	95	83
9		LDA	#'O	A9	4F	
B		JSR	WRITE_CHAR	20	6B	81
E		LDA	#'K	A9	4B	
8420		JSR	WRITE_CHAR	20	6B	81
8423		JMP	OUTPUTLN_TTY	4C	82	81

8426	TOPLEVEL:	JSR	WRITE_PROMPT	20	16	84
9		JSR	INPUT_TTY	20	95	81
C		JSR	RESET_LINE	20	95	83
F		JSR	PEEK_LINE	20	8F	83
32		CMP	#'.	C9	2E	
4		BNE	NEXT	D0	01	
6		RTS		60		
7	NEXT:	CMP	#'[	C9	5B	
9		BNE	NEXT	D0	08	
B		INC	LINE_OFF	E6	FE	
D		JSR	SYMBALIZE	20	64	84
40		JMP	TOPLEVEL	4C	26	84
3	NEXT:	CMP	#']	C9	5D	
5		BNE	NEXT	D0	08	
7		INC	LINE_OFF	E6	FE	
9		JSR	DEFINE_SYMBAL	20	DE	84
C		JMP	TOPLEVEL	4C	26	84
F	NEXT:	CMP	#'/'	C9	2F	
51		BNE	NEXT	D0	08	
3		INC	LINE_OFF	E6	FE	
5		JSR	REPLACE_REFS	20	36	84
8		JMP	TOPLEVEL	4C	26	84
B	NEXT:	JSR	SYMBALIZE	20	64	84
E		JSR	EMITBUF	20	00	C1
8461		JMP	TOPLEVEL	4C	26	84

8464	SYMBALIZE:	JSR	RESET_EMIT	20	B9	83
8467	LOOP:	JSR	SKIP_SPACE	20	9A	83
A		BCC	NOT_EOL	90	03	
C		JMP	EOL	4C	<u>C7</u>	<u>84</u>
F	NOT_EOL:	JSR	PEEK_LINE	20	8F	83
72		CMP	#'?	C9	3F	
4		BNE	NEXT	D0	<u>08</u>	
6		INC	LINE_OFF	E6	FE	
8		JSR	EMIT_IF	20	D5	83
B		JMP	LOOP	4C	67	84
E	NEXT:	CMP	#'!	C9	21	
80		BNE	NEXT	D0	<u>08</u>	
2		INC	LINE_OFF	E6	FE	
4		JSR	EMIT_REPEAT	20	E4	83
7		JMP	LOOP	4C	67	84
A	NEXT:	CMP	#'#	C9	23	
C		BNE	NEXT	D0	<u>0D</u>	
E		INC	LINE_OFF	E6	FE	
90		JSR	READ_BYTE	20	4D	81
3		STA	ARG_LO	85	0E	
5		JSR	EMIT_CONST	20	F3	83
8		JMP	LOOP	4C	67	84
B	NEXT:	CMP	#'<	C9	3C	
D		BNE	NEXT	D0	<u>0B</u>	
F		INC	LINE_OFF	E6	FE	
A1		JSR	READ_ARG	20	AE	83
4		JSR	EMIT_LOAD	20	02	84
84A7		JMP	LOOP	4C	67	84

84AA	NEXT:	CMP #'>	C9	3E	
C		BNE NEXT	D0	<u>0B</u>	
E		INC LINE_OFF	E6	FE	
80		JSR READ_ARG	20	AE	83
3		JSR EMIT_STORE	20	11	84
6		JMP LOOP	4C	67	84
9	NEXT:	JSR LOOKUP_SYMBAL	20	37	83
C		BCS ERROR	B0	<u>06</u>	
E		JSR EMIT_CALL	20	C6	83
C1		JMP LOOP	4C	67	84
4	ERROR:	JSR REPORT_ERROR	20	<u>CC</u>	<u>84</u>
84C7	EOL:	LDA #\$60	A9	60	
9		JMP EMIT_BYTE	4C	BE	83

84CC	REPORT_ERROR:	LDA LINE_OFF	A5	FE	
E		PHA	48		
F		JSR RESET_LINE	20	95	83
D2		LDA #'?	A9	3F	
4		JSR WRITE_CHAR	20	6B	81
7		PLA	68		
8		JSR WRITE_BYTE	20	5C	81
84DB		JMP OUTPUTLN_TTY	4C	82	81

84DE	DEFINE_SYMBOL:	JSR	LOOKUP_SYMBOL	20	37	83
E1		BCS	NOT_FOUND	B0	03	
3		JMP	REPORT_ERROR	4C	CC	84
6	NOT_FOUND:	LDA	#\$00	A9	00	
8		STA	TEMP_LENGTH	85	FD	
84EA	LOOP:	JSR	PEEK_LINE	20	8F	83
D		CMP	#\$0D	C9	0D	
F		BEQ	SCANNED	F0	<u>0E</u>	
F1		CMP	#\$20	C9	20	
3		BEQ	SCANNED	F0	<u>0A</u>	
5		JSR	WRITE_STAB	20	30	83
8		INC	LINE_OFF	E6	FE	
A		INC	TEMP_LENGTH	E6	FD	
C		JMP	LOOP	4C	<u>EA</u>	<u>84</u>
F	SCANNED:	LDA	TEMP_LENGTH	A5	FD	
8501		ORA	#\$80	09	80	
3		JSR	WRITE_STAB	20	30	83
6		LDA	EMIT_OFF	A5	10	
8		JSR	WRITE_STAB	20	30	83
B		LDX	#\$00	A2	00	
D	COPY_LOOP:	LDA	EMITBUF, X	BD	00	C1
10		JSR	WRITE_STAB	20	30	83
3		INX		E8		
4		CPX	EMIT_OFF	E4	10	
6		BNE	COPY_LOOP	D0	<u>F5</u>	
8		LDA	#\$00	A9	00	
A		JSR	WRITE_STAB	20	30	83
851D		JMP	RESET_STAB_PTR	4C	00	83

...

All written and hand-assembled!

At this point it's handy to have a list of the entry points, so I wrote that up too:

ENTRY POINTS IN SITU-SOL

RESET_STAB_PTR	\$8300	
INCR_STAB_PTR	8309	ADD_A_STAB_PTR
ADD_TO_STAB_PTR	830D	← 830B
SUB_STAB_PTR	831B	
PEEK_STAB	8329	
WRITE_STAB	8330	
LOOKUP_SYMBAL	8337	
MATCH_SYMBAL	834D	
PEEK_LINE	838F	
RESET_LINE	8395	
SKIP_SPACE	839A	
READ_ARG	83AE	
RESET_EMIT	83B9	
EMIT_BYTE	83BE	
EMIT_CALL	83C6	
EMIT_IF	83D5	
EMIT_REPEAT	83E4	
EMIT_CONST	83F3	
EMIT_LOAD	8402	
EMIT_ARG	8404	
EMIT_STORE	8411	

WRITE_PROMPT	\$ 8416
TOPLEVEL	8426
SYMBALIZE	8464
REPORT_ERROR	84CC
DEFINE_SYMBAL	84DE

TEMP_LENGTH	\$ FD	(TMP_A in SITU-MON)
-------------	-------	---------------------

STAB_PTR[_LO]	\$ 0A
STAB_PTR_HI	0B
STAB_DELTA	0C
STAB_BYTE	0D
ARG_LO	0E
ARG_HI	0F
EMIT_OFF	10

EMITBUF	\$ C100
STAB_BASE	\$ 0900

Now, to enter it... luckily, entering hex digits with SITU-MON is a lot easier than entering bits using front-panel switches.



VICE: C64 emulator

```
0000
A8300
8300
WA900850AA909850B60
8309
WA901850C
830D
W18A50A650C850AA50B6900850B60
831B
W38A50AE50C850AA50BE900850B60
8329
WA000B10A850D60
8330
WA000910A4C0983
8337
W200083202983F00C204D83B0F6A902
8346
W200D8318603860
834D
█
```



VICE: C64 emulator

```
8330
WA000910A4C0983
8337
W200083202983F00C204D83B0F6A902
8346
W200D8318603860
834D
WA6FE2029832980D00EA50DD00C0D012200983
8360
WE84C4F83BD00C0C920F011C90DF00D
836F
W2029832980D00A209834C6F8386FE1860
8380
W200983202983850CE60C200D833860
838F
WA0FEB900C060
8395
WA90085FE60
839A
W208F83C90DF006C920F00418603860E6FE4C9A8
3
83AE
W204D81850F204D81850E60
83B9
■
```



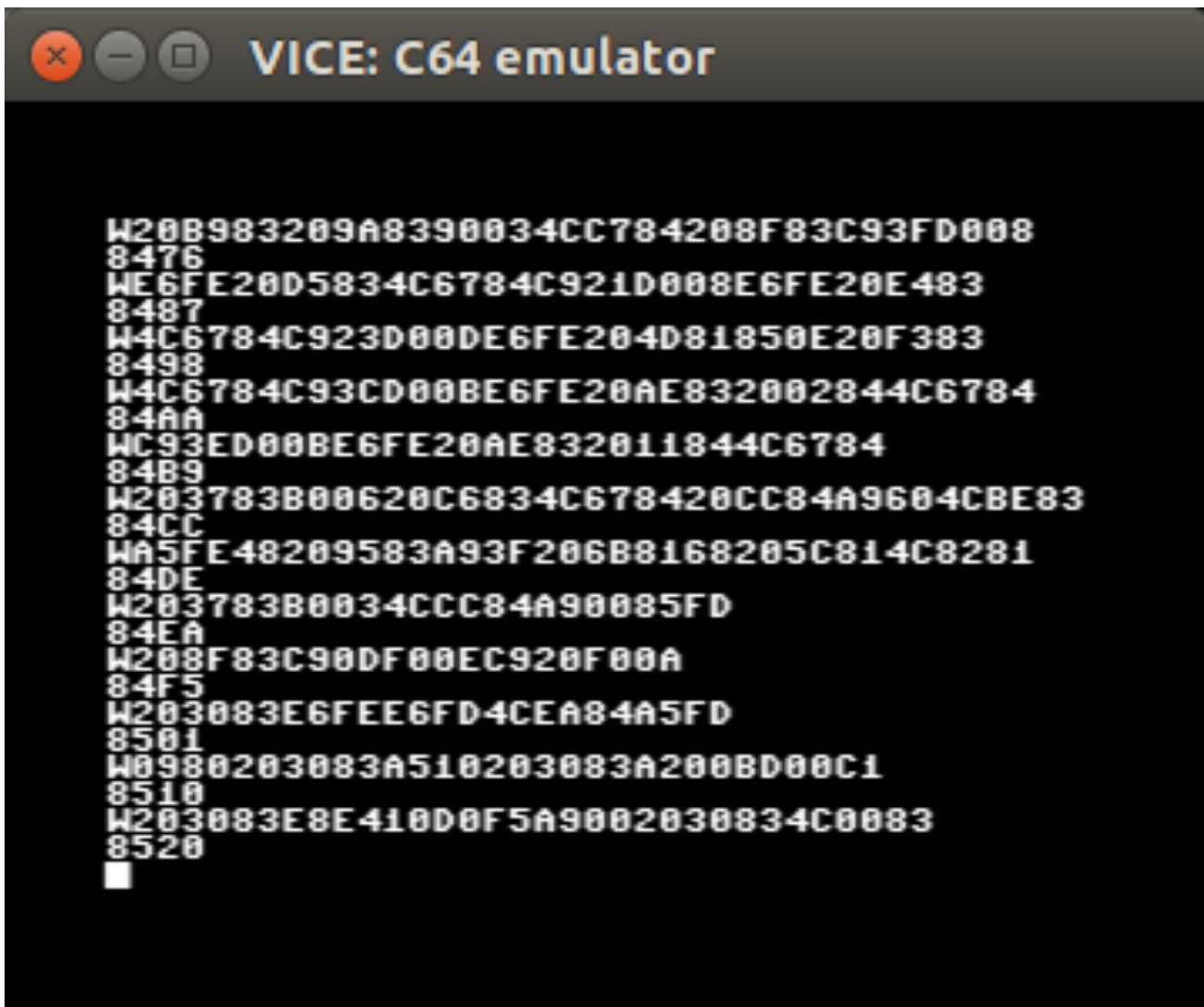
VICE: C64 emulator

```
839A
W208F83C90DF006C920F00418603860E6FE4C9A8
3
83AE
W204D81850F204D81850E60
83B9
WA900851060
83BE
WA4109900C1E61060
83C6
WA92020BE83A50A20BE83A50B4CBE83
83D5
WA9EA20BE83A9F020BE83A9034CBE83
83E4
WA9D020BE83A9FA20BE83A9EA4CBE83
83F3
WA9A920BE83A50E20BE83A9EA4CBE83
8402
WA9AD20BE83A50E20BE83A50F4CBE83
8411
WA98D4C0484
8416
W209583A94F206B81A94B206B814C8281
8426
█
```



VICE: C64 emulator

```
WA9AD20BE83A50E20BE83A50F4CBE83
8411
WA98D4C0484
8416
W209583A94F206B81A94B206B814C8281
8426
W201684209581209583208F83C92ED00160
8437
WC95BD008E6FE2064844C2684
8443
WC95DD008E6FE20DE844C2684
844F
WC92FD008E6FE2036844C2684
845B
W2064842000C14C2684
8464
W20B983209A8390034CC784208F83C93FD008
8476
WE6FE20D5834C6784C921D008E6FE20E483
8487
W4C6784C923D00DE6FE204D81850E20F383
8498
W4C6784C93CD00BE6FE20AE832002844C6784
84AA
■
```

The image shows a screenshot of the VICE: C64 emulator window. The title bar at the top reads "VICE: C64 emulator" and includes standard window control buttons (close, minimize, maximize). The main area of the window is black and displays a memory dump in white text. The dump consists of 16 lines, each showing a hexadecimal address followed by a hexadecimal value. The addresses are: 8476, 8487, 8498, 84AA, 84B9, 84CC, 84DE, 84EA, 84F5, 8501, 8510, and 8520. The values are: W20B983209A8390034CC784208F83C93FD008, WE6FE20D5834C6784C921D008E6FE20E483, W4C6784C923D00DE6FE204D81850E20F383, W4C6784C93CD00BE6FE20AE832002844C6784, WC93ED00BE6FE20AE832011844C6784, W203783B00620C6834C678420CC84A9604CBE83, WA5FE48209583A93F206B8168205C814C8281, W203783B0034CCC84A90085FD, W208F83C90DF00EC920F00A, W203083E6FEE6FD4CEA84A5FD, W0980203083A510203083A200BD00C1, and W203083E8E410D0F5A9002030834C0083. A small white cursor is visible at the bottom left of the dump area.

```
W20B983209A8390034CC784208F83C93FD008
8476
WE6FE20D5834C6784C921D008E6FE20E483
8487
W4C6784C923D00DE6FE204D81850E20F383
8498
W4C6784C93CD00BE6FE20AE832002844C6784
84AA
WC93ED00BE6FE20AE832011844C6784
84B9
W203783B00620C6834C678420CC84A9604CBE83
84CC
WA5FE48209583A93F206B8168205C814C8281
84DE
W203783B0034CCC84A90085FD
84EA
W208F83C90DF00EC920F00A
84F5
W203083E6FEE6FD4CEA84A5FD
8501
W0980203083A510203083A200BD00C1
8510
W203083E8E410D0F5A9002030834C0083
8520
█
```

All done!

Of course, I made several errors.

I caught one error while reviewing what I had entered with the M command. I had simply mistyped a line. That one was easy to fix.

Other errors - I had to test the program first.

And in order to test it, I needed to have a symbol table for it to work on. I made one up with two very simple “commands” whose effect would be both obvious to see, and non-intrusive on the rest of the system. I decided to have them call two of the simpler routines present in SITU-MON (WRITE_PROMPT and INCR_ADDR). (And yes, you’re right, they should really be called \$ADDR and \$INC, with the dollar signs – thank you for paying attention to the spec. But, this

was just to test...)

DEMO SYMBOL TABLE

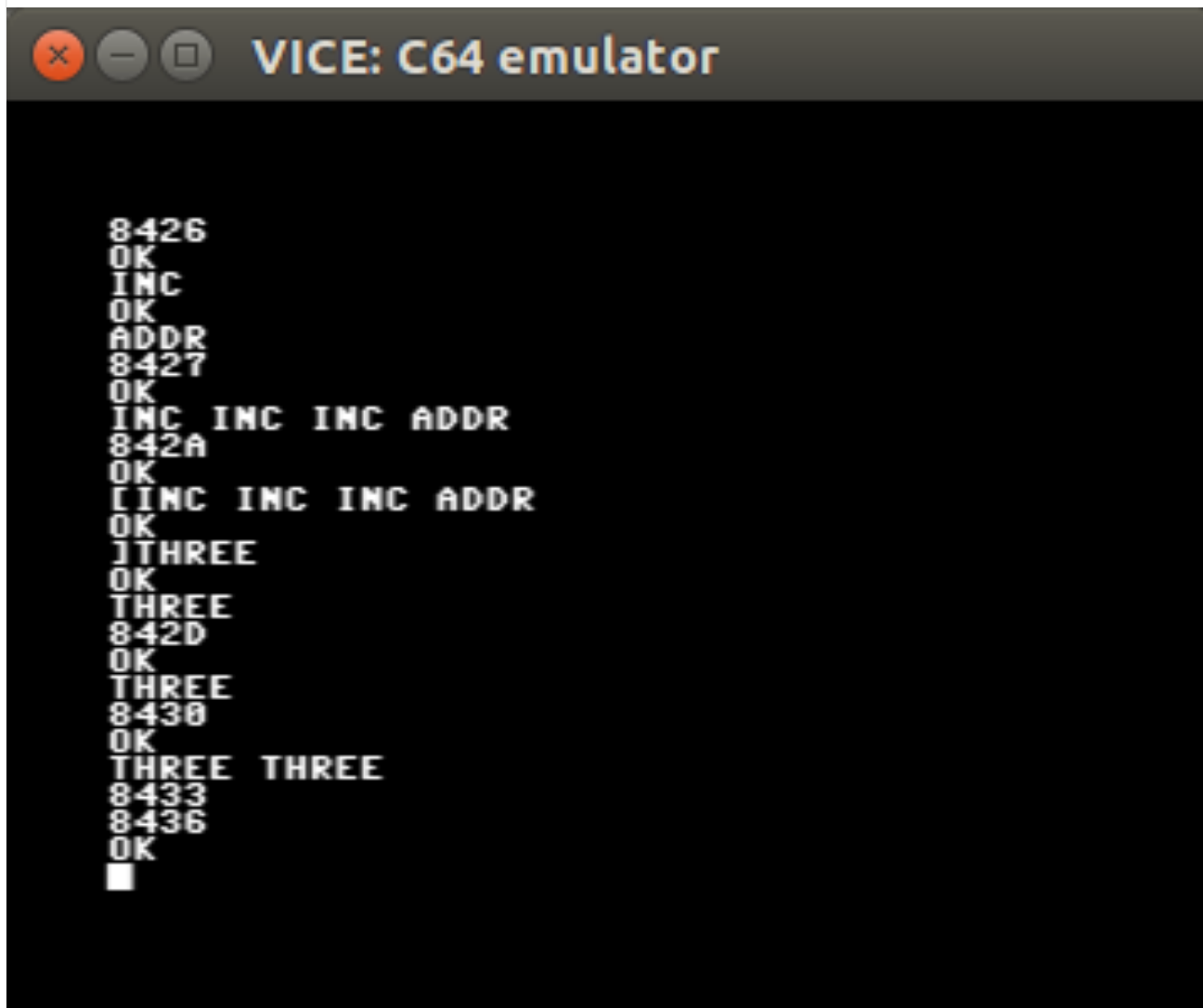
\$0900	"ADDR" #84 #04	41 44 44 52 84 04
6	JSR WRITE_PROMPT	20 74 81
9	RTS	60
\$090A	"INC" #83 #04	49 4E 43 83 04
F	JSR INCR_ADDR	20 DC 81
12	RTS	60
3	#000	00

TEST DRIVER

8200	JSR INPUT_TTY	20 95 81
	JSR ~~SYM~~ RESET_LINE	20 95 83
	JSR SYMBALIZE	20 64 84
	RTS	60

ADD_A_STAB_PTR just before ADD_TO_STAB_PTR, and call that one instead.

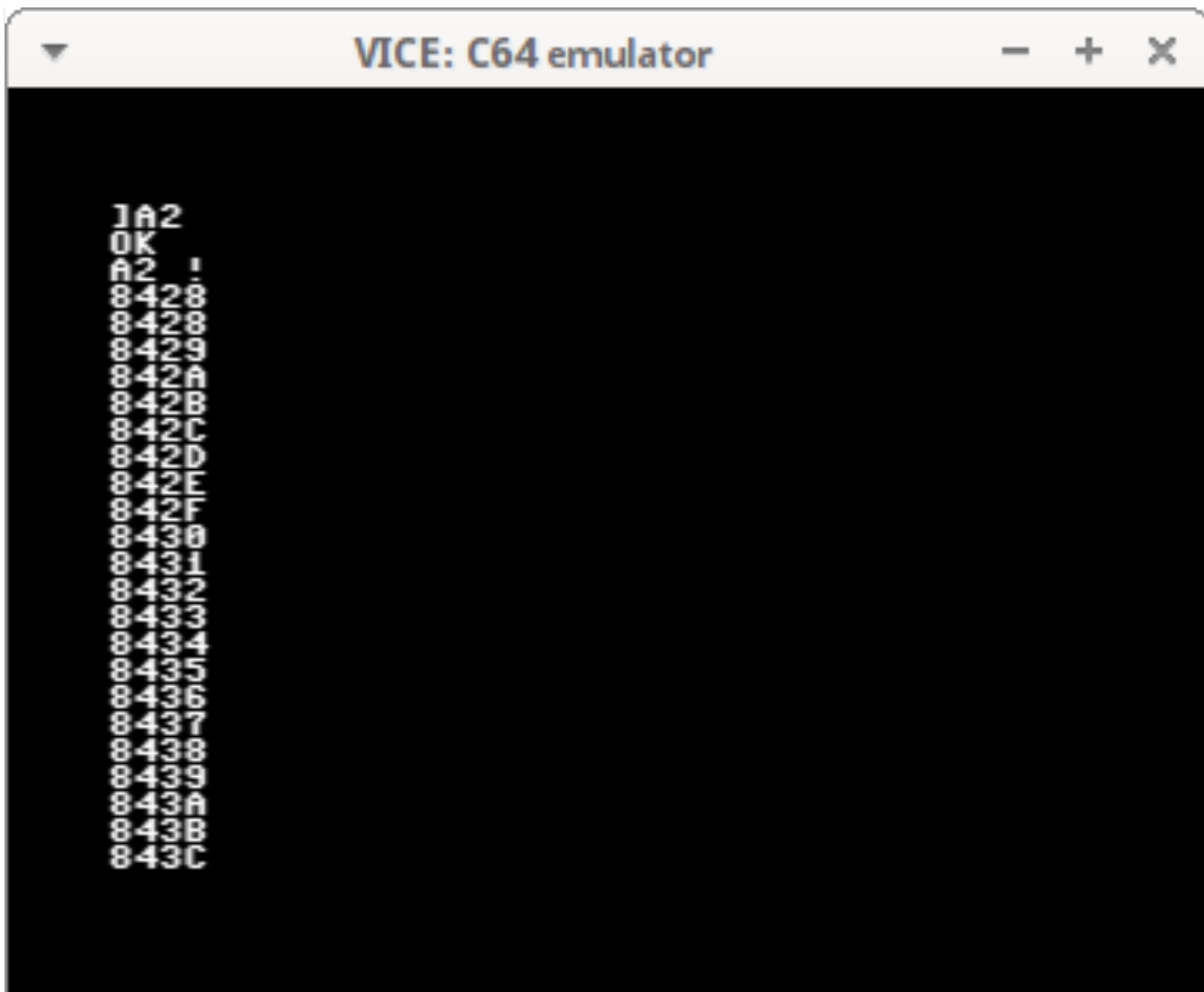
Once I fixed that, I was able to compile simple “programs”:

A screenshot of the VICE: C64 emulator window. The title bar shows standard window controls (close, minimize, maximize) and the text "VICE: C64 emulator". The main window area has a black background with white text. The text is organized into two columns: memory addresses on the left and assembly instructions on the right. The addresses are 8426, 8427, 842A, 842D, 8430, 8433, and 8436. The instructions are OK, INC, ADDR, INC INC INC ADDR, JTHREE, THREE, and THREE THREE. At the bottom, there is a small white square cursor.

```
8426
OK
INC
OK
ADDR
8427
OK
INC INC INC ADDR
842A
OK
INC INC INC ADDR
OK
JTHREE
OK
THREE
842D
OK
THREE
8430
OK
THREE THREE
8433
8436
OK
█
```

There was still one more bug. The code generated when compiling a ! (repeat) was wrong – it was jumping one byte too far back in memory. Once spotted, that was another easy fix. And then...

```
A83E9
83E9
WA9FB
83EB
A8426
8426
G
OK
#0 ? ADDR
OK
#1 ? ADDR
8426
OK
[INC ADDR #0
OK
JA1
OK
A1 !
8427
OK
[INC ADDR #1
OK
JA2
OK
A2 !
```



Huzzah!

A moderate success. We have, on this emulated computer, a programming environment, hand-written and hand-assembled and hand-entered in hexadecimal (using a less sophisticated programming environment, which was itself hand-written and hand-assembled and hand-entered in binary using simulated front-panel switches.)

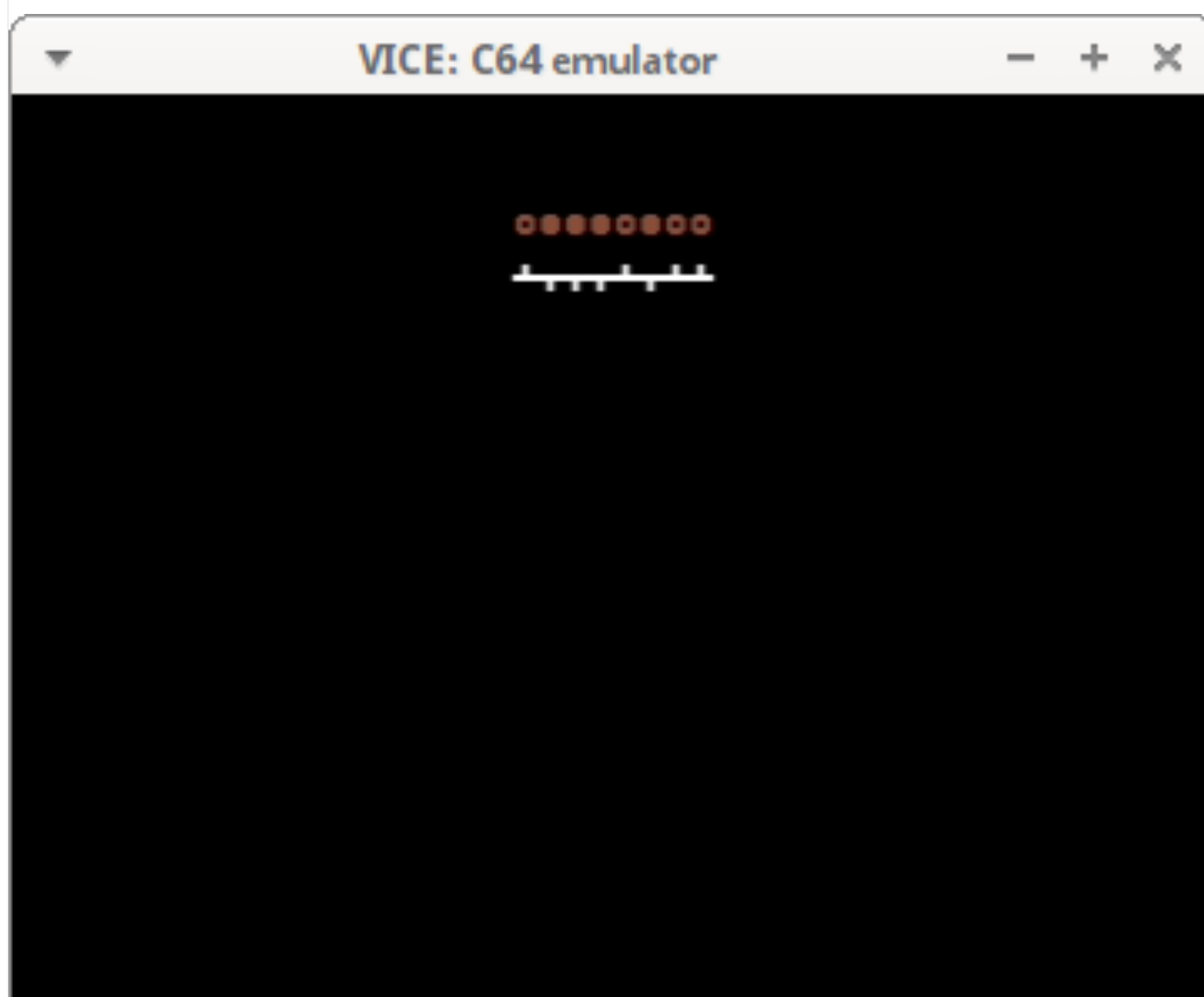
But why do I consider this a moderate success rather than an unqualified success? Well, I'll write a retrospective blog post on that topic shortly; stay tuned! :)

1 note

I love bits. Bits are great when you're coding and you want 2,000 of something

(with apologies to Mitch Hedberg for the title of this post)

Right! So almost as soon as I finished writing the code for SITU-MON I started entering it into the emulated Commodore 64, using the simulated front-panel switches. At first this was in dribs and drabs, a few bytes here, a few bytes there.



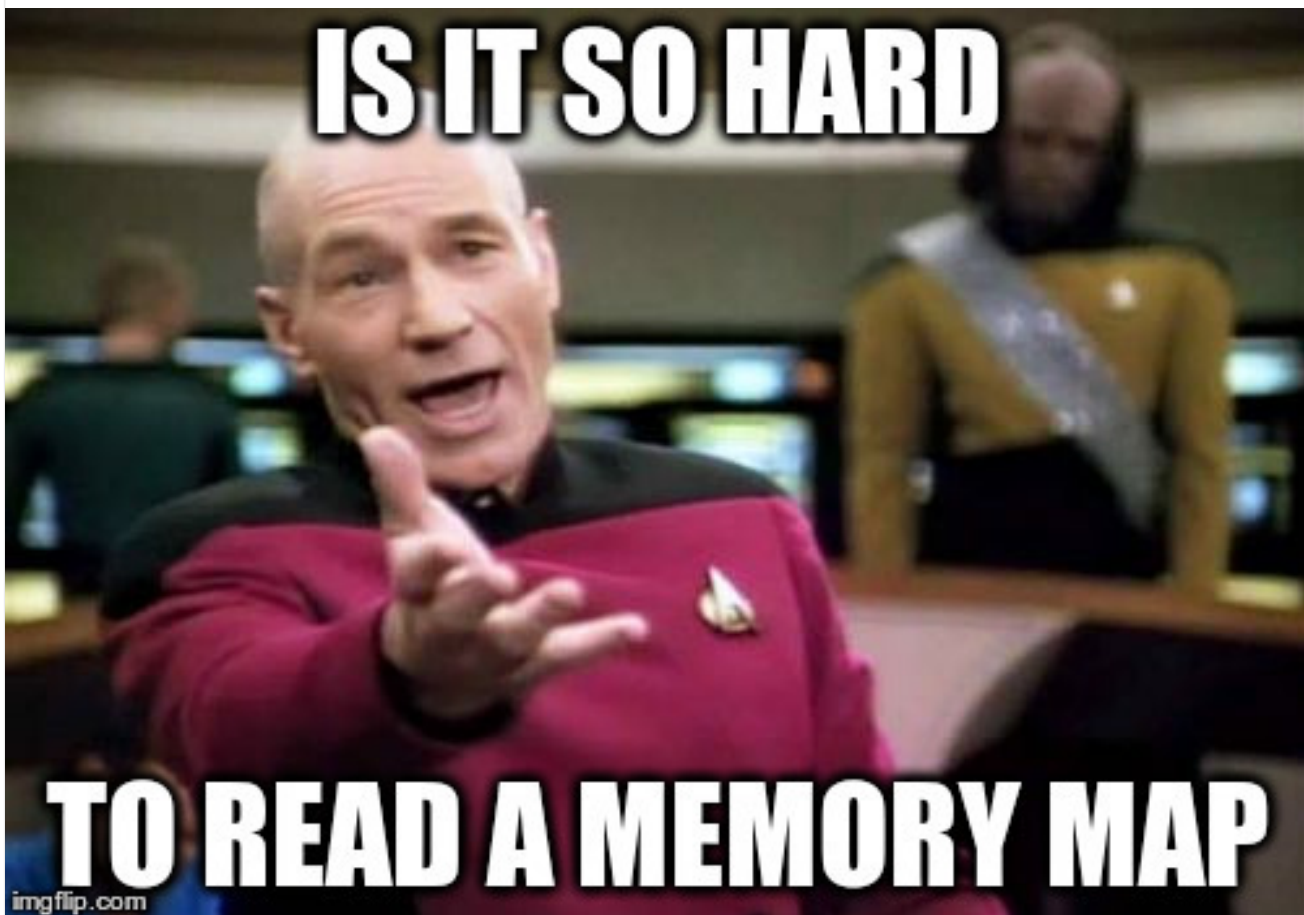
I won't bore you with thousands of screenshots; it all looked basically as described in a better front-panel simulator. (And for what it would look like in the physical world, watch the video on the loading the code post.) It's not difficult, exactly, it's just tedious and fatiguing.

Finally, I had entered all 2,000 bits, and I reset the address to the beginning, planning to review the bits I had entered to confirm they were the same bits that were written on the paper.

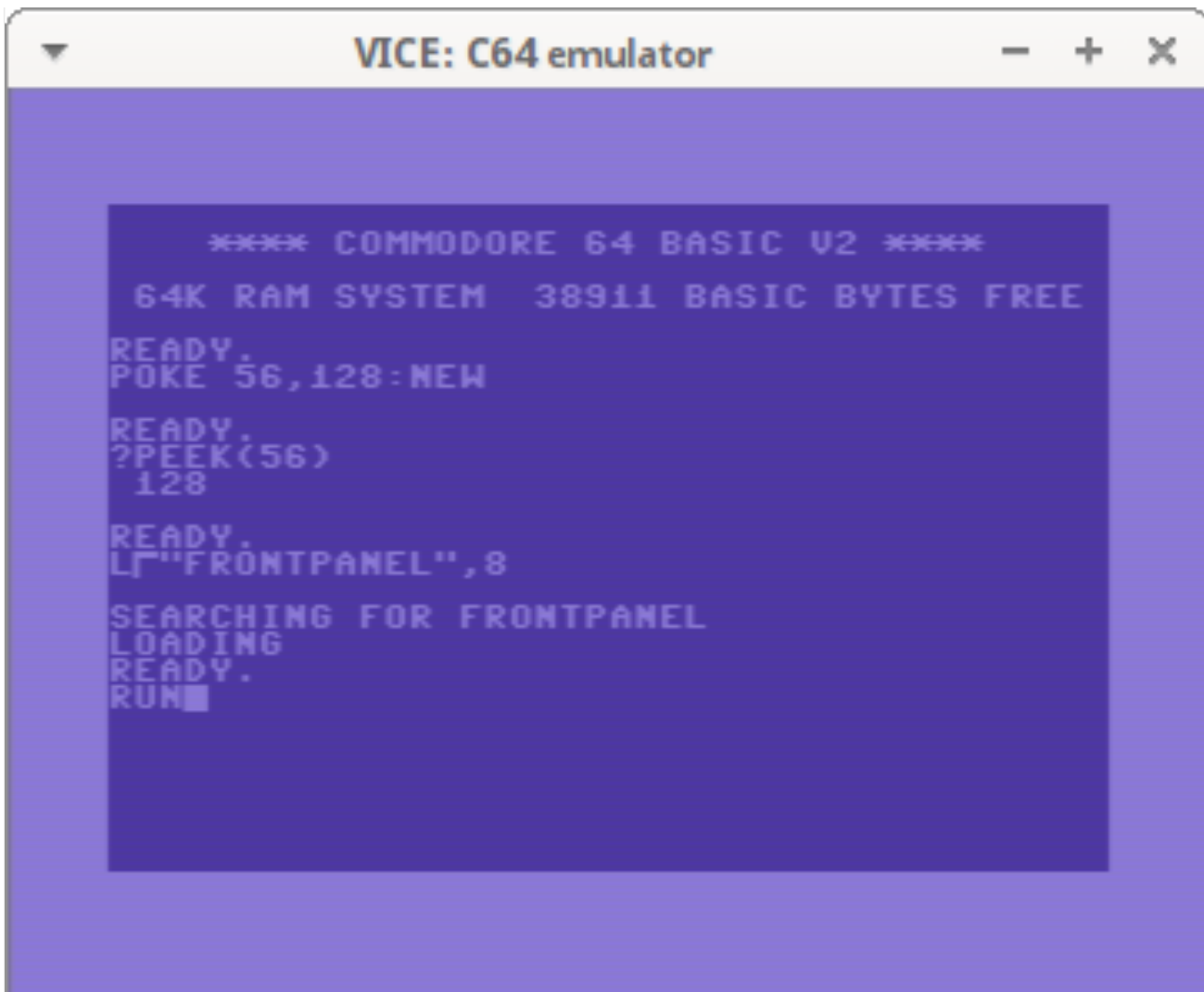
To my considerable surprise, they were completely and utterly different bits than on the paper. They were nothing like what I entered.

Then I realized what had happened. The SITU-MON code starts at \$8100, but this is inside BASIC's default memory range, which extends up to \$9FFF. Also, the improved front-panel simulator creates a lot of temporary strings compared to the first version.

These grew downward in memory and completely clobbered the 250 bytes I had entered!



So I lowered the top of BASIC memory, and started all over again.



The second time doing this, at least, I was able to enter the bits much more quickly; I had had practice, and had a block of time to myself that I could dedicate to it. And when I reset the address to review the code, it was, in fact, the code I entered.

Well, mostly. I skipped a byte about 80% through. So, I re-entered the last 20% and reviewed it again. In the second review, I found one wrong byte, but thankfully it was all in sequence and I didn't have to shift anything.

Satisfied that what was in the computer matched what was written down on paper, I reset the address to the beginning again and... executed!

And nothing happened. It just hung.

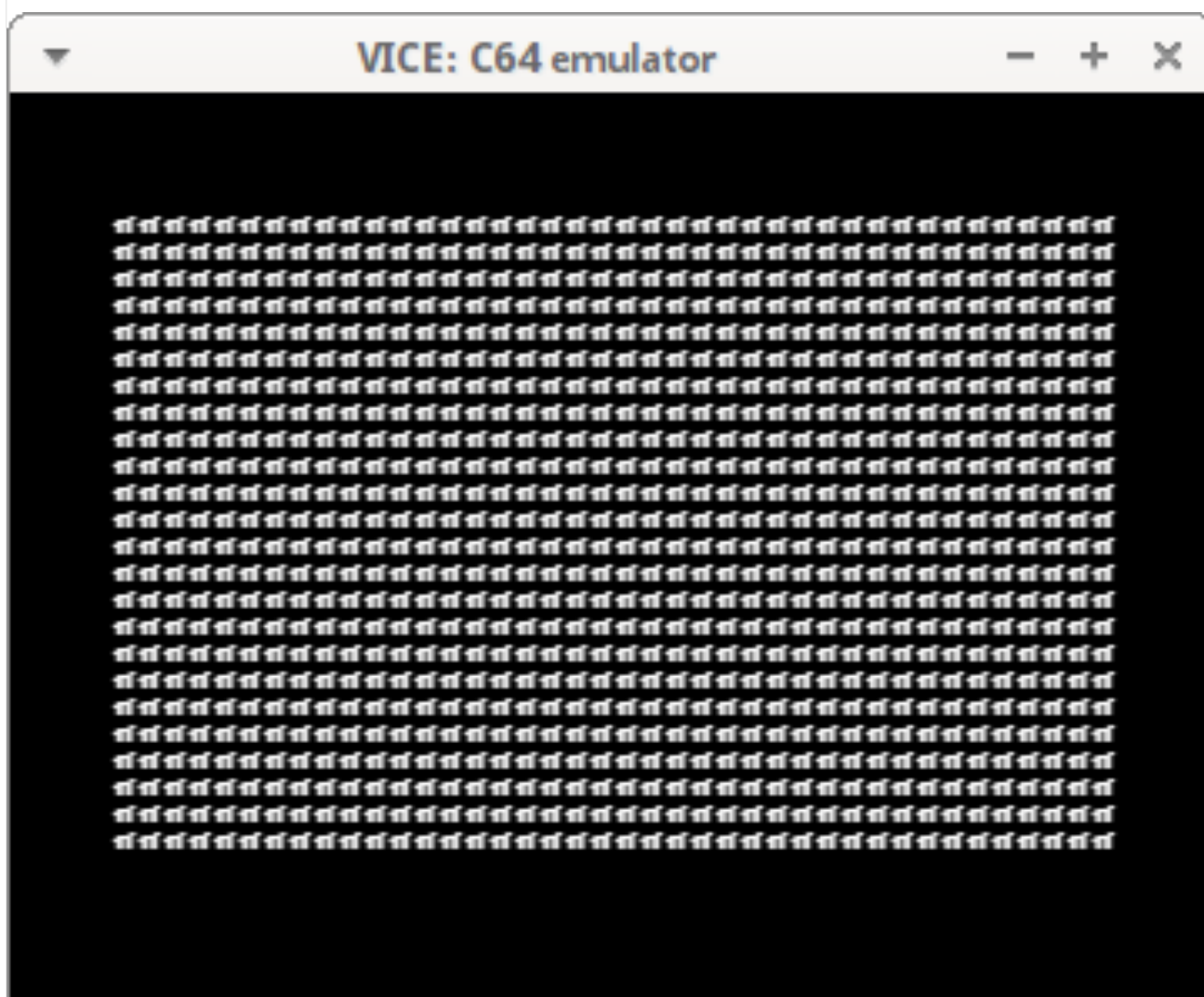
Thus, the debugging began.

I pressed RUN/STOP+RESTORE and re-RUN the front-panel simulator – the moral equivalent of pressing a Reset button on a single-board computer which resets the CPU to a known state without affecting the contents of memory.

But how to debug something like this?

I looked back to the listing on paper and traced the execution path through it, as best I could, by hand. I couldn't see where there was a problem.

The first routine the code calls is WRITE_PROMPT. In the front-panel simulator, I advanced to the WRITE_PROMPT routine and executed it. And I got this:



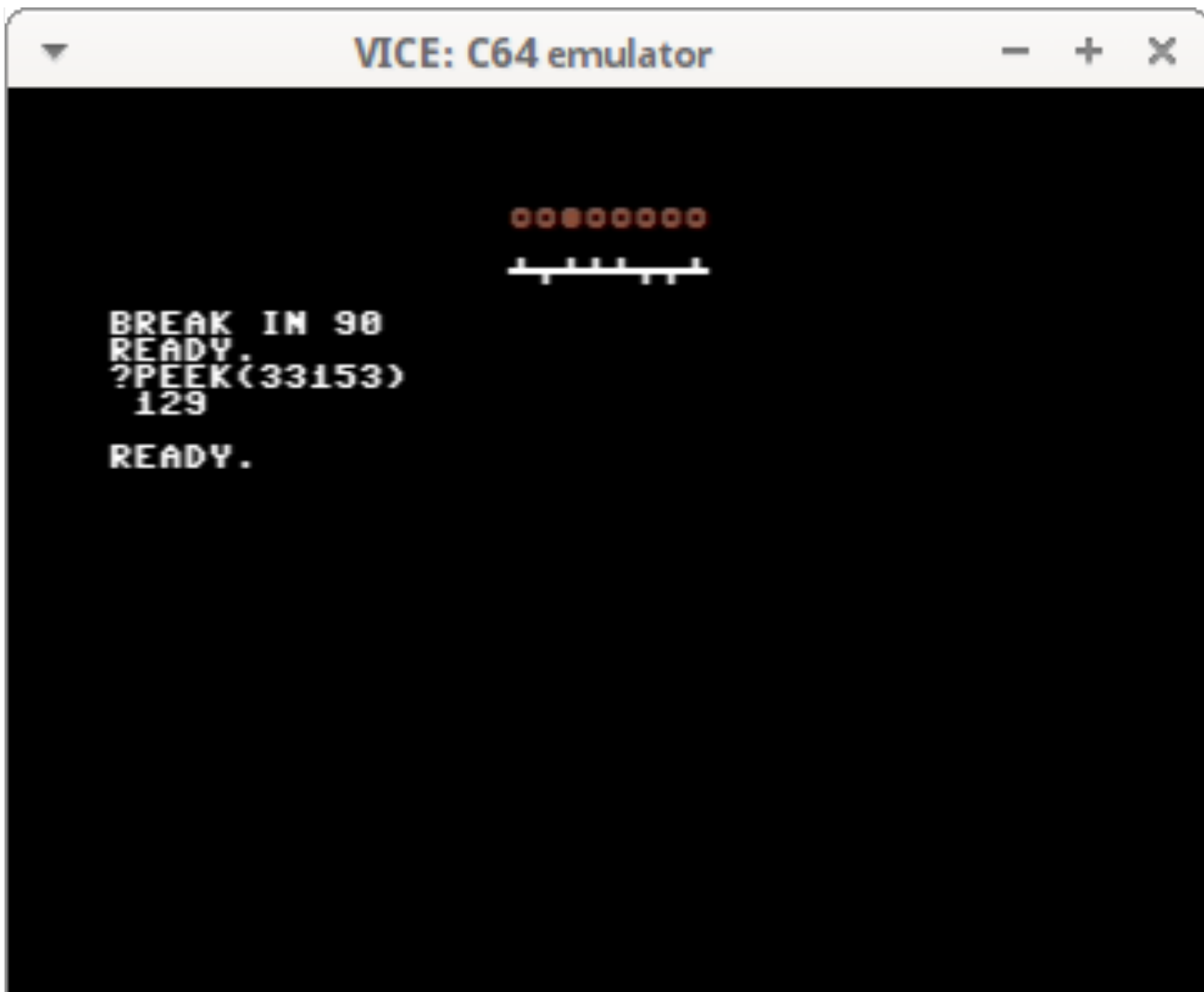
Yeah, no, that definitely wasn't in the spec.

A mild panic set in. The bug could be *anywhere*. I might've written down the wrong binary for a hex value; I might've written down the wrong hex value for an opcode, or I might've made a programming error in the code itself.

I had just finished entering 2,000 bits on a simulated front-panel – for the second time! – and I was in no mood to spend the rest of the weekend, or possibly the rest of the RetroChallenge, poring over my program.

So I took a bit of a mulligan. It was a small thing, and I could rationalize it by saying *“let's pretend there are 16 green LEDs on the front panel for the address lines”* or something. But still, I'm not proud of it, and it should count against my RetroChallenge score (however vague and nebulous a concept that may be.)

I converted the address of the OUTPUTLN_TTY routine to decimal (33153), dropped to BASIC, and PEEK'ed that address to see what byte was there. The first instruction of OUTPUTLN_TTY is LDA #\$00, so I was expecting to see 169 (\$A9), but...



Noticing that the byte just *before* OUTPUTLN_TTY was \$81, it occurred to me: maybe I made a mistake assigning addresses to labels.

So I read over the code again, and, indeed.

8139	RD. HX.	JSR \$8130		20	30	81
813B	PS. HX.	LDY #00		A0	00	

See the problem?

So! A bit painful that this mis-numbering happened so early in the code, but, there was nothing to be done except to go through it all and correct the addresses:

\$8100	MAIN	JSR \$8173 74	20 79 81	00000000	01110000	10000001
8103		JSR \$8194 95	20 95 81	00100000	10010100	10000001
8106		LDA #01	A9 01	10101001	00000001	
8108		STA \$FE	85 FE	10000101	11111110	
810A		JSR \$8110	20 10 81	00100000	00010000	10000001
810D		JMP \$8100	4C 00 81	01001100	00000000	10000001
8110	DISP	LDA \$C000	AD 00 C0	10101101	00000000	11000000
8113		CMP #041	C9 41	11001001	01000001	
8115		BNE \$03	D0 03	11010000	00000011	
8117		JMP \$81A7 A8	4C A8 81	01001100	10100000	10000001
811A		CMP #04D	C9 4D	11001001	01001101	
811C		BNE \$03	D0 03	11010000	00000011	
811E		JMP \$81B2 B3	4C B3 81	01001100	10110010	10000001
8121		CMP #057	C9 57	11001001	01010111	
8123		BNE \$03	D0 03	11010000	00000011	
8125		JMP \$81C4 C5	4C C5 81	01001100	11000100	10000001
8128		CMP #047	C9 47	11001001	01000111	
812A		BNE \$03	D0 03	11010000	00000011	
812C		JMP (\$00FB)	6C FB 00	01101100	11110101	00000000
812F		RTS	60	01100000		
8130	RD. CH.	LDY \$FE	A4 FE	10100100	11111110	
8132		LDA \$C000, Y	B9 00 C0	10111001	00000000	11000000
8135		INY	C8	11001000		
8136		STY \$FE	84 FE	10000100	11111110	
8138		RTS	60	01100000		
8139	RD. HX.	JSR \$8130	20 30 81	00100000	00110000	10000001
813C	PS. HX.	LDY #000	A0 00	10100000	00000000	
813E		CMP \$81E8, Y	D9 E8 81	11011001	11101000	10000001
8141		BNE \$02	D0 02	11010000	00000010	

8143	8142	TYA	98	10011000		
8144	8143	RTS	60	01100000		
5	8144	INY	C8	11001000		
6	8145	CPY #010	C0 10	11000000	00010000	
8	8147	BNE \$F4	D0 F4	11010000	11110100	
A	8149	LDA #000	A9 00	10101001	00000000	
C	814B	RTS	60	01100000		
814D	814C	RD. BY. JSR \$8139	20 39 81	00100000	00111001	10000001
8150	814F	ASL A	0A	00001010		
1	8150	ASL A	0A	00001010		
2	8151	ASL A	0A	00001010		
3	8152	ASL A	0A	00001010		
4	8153	STA \$FD	85 FD	10000101	11111101	
6	8155	JSR \$8139	20 39 81	00100000	00111001	10000001
9	8158	ORA \$FD	05 FD	00000101	11111101	
B	815A	RTS	60	01100000		
815C	8158	WR. BY. PHA	48	01001000		
D	815C	LSR A	4A	01000000	01001010	
E	815D	LSR A	4A	01000000	01001010	
F	815E	LSR A	4A	01000000	01001010	
8160	815F	LSR A	4A	01000000	01001010	
1	8160	JSR \$8164 C5	20 65 81	00100000	01100100	10000001
4	8163	PLA	68	01101000		
8165	8164	WR. HX. AND #00F	29 0F	00101001	00001111	
7	8166	TAY	A8	10101000		
8	8167	LDA \$81E8, Y	B9 E8 81	10111001	11101000	10000001
816B	816A	WR. CH. LDY \$FE	A4 FE	10100100	11111110	
D	816C	STA \$C000, Y	99 00 C0	10011001	00000000	11000000
8170	816F	INY	C8	11001000		
1	8170	STY \$FE	84 FE	10000100	11111110	
3	8172	RTS	60	01100000		

8174	\$ 8173	WR.PR.	LDA #500	A9 00	10101001	00000000	
6	8175		STA \$FE	85 FE	10000101	11111110	
8	8177		LDA \$FC	A5 FC	10100101	11111100	
A	8179		JSR \$815B SC	20 5B 81	00100000	01011000	10000001
D	817C		LDA \$FB	A5 FB	10100101	11111011	
F	817E		JSR \$815B SC	20 5B 81	00100000	01011010	10000001
8182	8181	O.P.LN	LDA #500	A9 00	10101001	00000101	
4	8183		JSR \$816A CB	20 6A 81	00100000	01101010	10000001
8187	8186	O.P.TTY	LDY #500	A0 00	10100000	00000000	
9	8188		LDA \$C000, Y	B9 00 C0	10111001	00000000	11000000
C	818B		JSR \$FFD2	20 D2 FF	00100000	11010010	11111111
F	818E		INY	C8	11001000		
8190	818F		CMP #500	C9 00	11001001	00001101	
2	8191		BNE \$F5	D0 F5	11010000	11110101	
4	8193		RTS	60	01100000		
8195	8194	IMP.TTY	LDY #500	A0 00	10100000	00000000	
7	8196		JSR \$FFCF	20 CF FF	00100000	11001111	11111111
A	8199		STA \$C000, Y	99 00 C0	10011001	00000000	11000000
D	819C		INY	C8	11001000		
E	819D		CMP #500	C9 00	11001001	00001101	
81A0	819F		BNE \$F5	D0 F5	11010000	11110101	
2	81A1		STA \$C000, Y	99 00 C0	10011001	00000000	11000000
5	81A4		JMP \$FFD2	4C D2 FF	01001100	11010010	11111111
81A8	81A7	SET AD.	JSR \$814C 40	20 4C 81	00100000	01001100	10000001
8	81AA		STA \$FC	85 FC	10000101	11111100	
D	81AC		JSR \$814C 40	20 4C 81	00100000	01001100	10000001
81B0	81AF		STA \$FB	85 FB	10000101	11111011	
2	81B1		RTS	60	01100000		
81B3	81B2	RD. MEM	LDX #510	A2 10	10100010	00010000	
5	81B4		LDY #500	A0 00	10100000	00000000	

81B7	\$ 81B6		LDA (\$FB), Y	B1 FB	10110001	11111011	
9	81B8		JSR \$815B SC	20 5B 81	00100000	01011000	10000001
C	81BB		JSR \$81DB DC	20 DB 81	00100000	11011000	10000001
F	81BE		DEX	CA	11001010		
81C0	81BF		BNE \$F3	D0 F3	11010000	11110011	
2	81C1		JMP \$818F 82	4C 82 81	01001100	10000040	10000001
81C5	81C4	WR. MEM	LDY \$FE	A4 FE	10100100	11111110	
7	81C6		LDA \$C000, Y	B9 00 C0	10111001	00000000	11000000
A	81C9		CMP #500	C9 00	11001001	00001101	
C	81CB		BNE \$01	D0 01	11010000	00000001	
E	81CD		RTS	60	01100000		
F	81CE		JSR \$814C 40	20 4C 81	00100000	01001100	10000001
81D2	81D1		LDY #500	A0 00	10100000	00000000	
4	81D3		STA (\$FB), Y	91 FB	10010001	11111011	
6	81D5		JSR \$81DB DC	20 DB 81	00100000	11011000	10000001
9	81D8		JMP \$81C4 C5	4C C5 81	01001100	11000100	10000001
81D C	81D8	INCR	CLC	18	00011000		
D	81DC		LDA \$FB	A5 FB	10100101	11111011	
F	81DE		ADC #501	C9 01	01101001	00000001	
81E1	81E0		STA \$FB	85 FB	10000101	11111011	
3	81E2		LDA \$FC	A5 FC	10100101	11111100	
5	81E4		ADC #500	C9 00	01101001	00000000	
7	81E6		STA \$FC	85 FC	10000101	11111100	
9	81E8		RTS	60	01100000		
81EA	81E9	HEXDIS	\$30	30	00110000		00111001
			\$31	31	00110000		01000001
			\$32	32	00110001		01000010
			\$33	33	00110010		01000011
			\$34	34	00110011		01000100
			\$35	35	00110100		01000101
			\$36	36	00110101		01000110
			\$37	37	00110111		
			\$38	38	00111000		

39

41

42

43

44

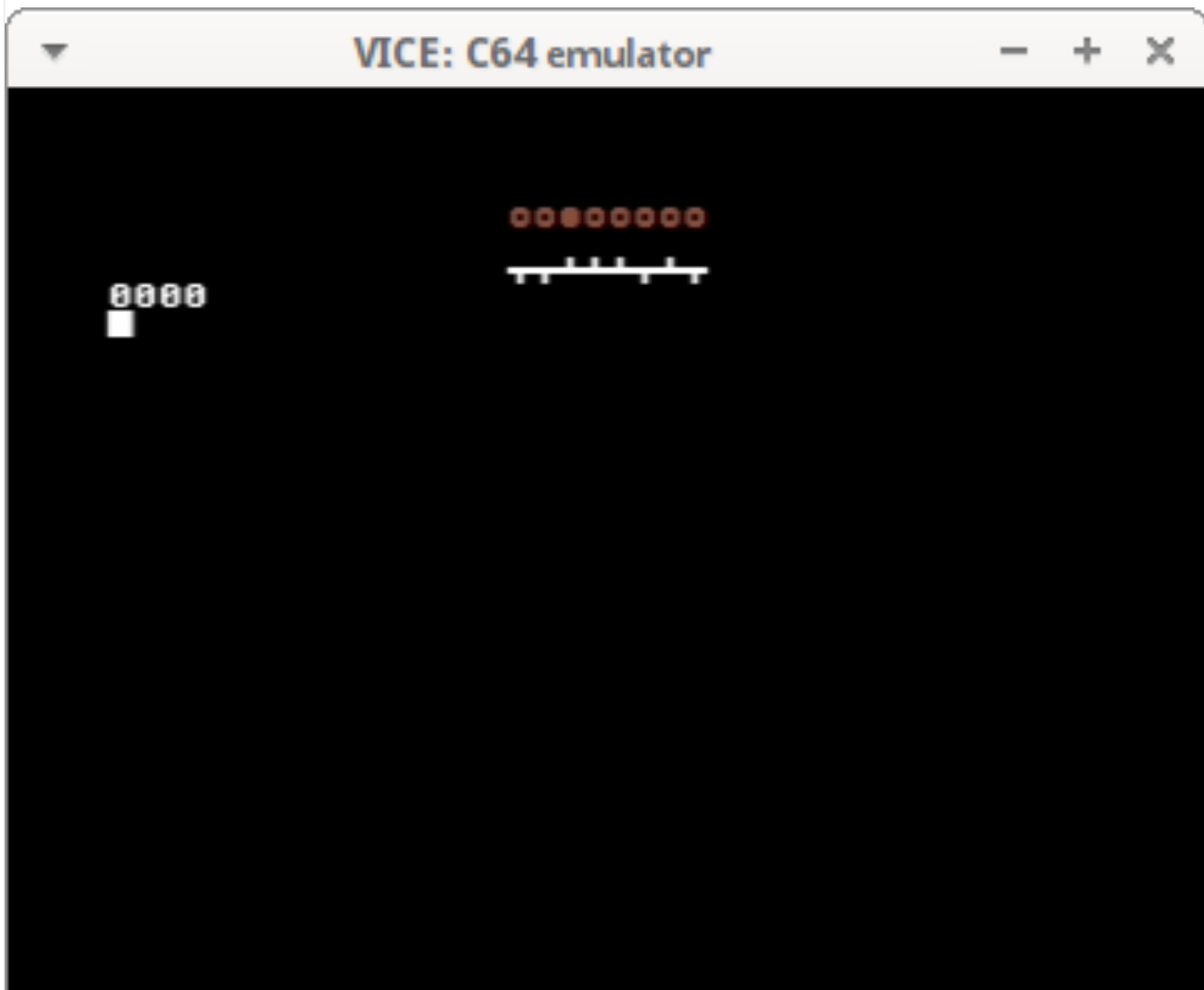
45

46

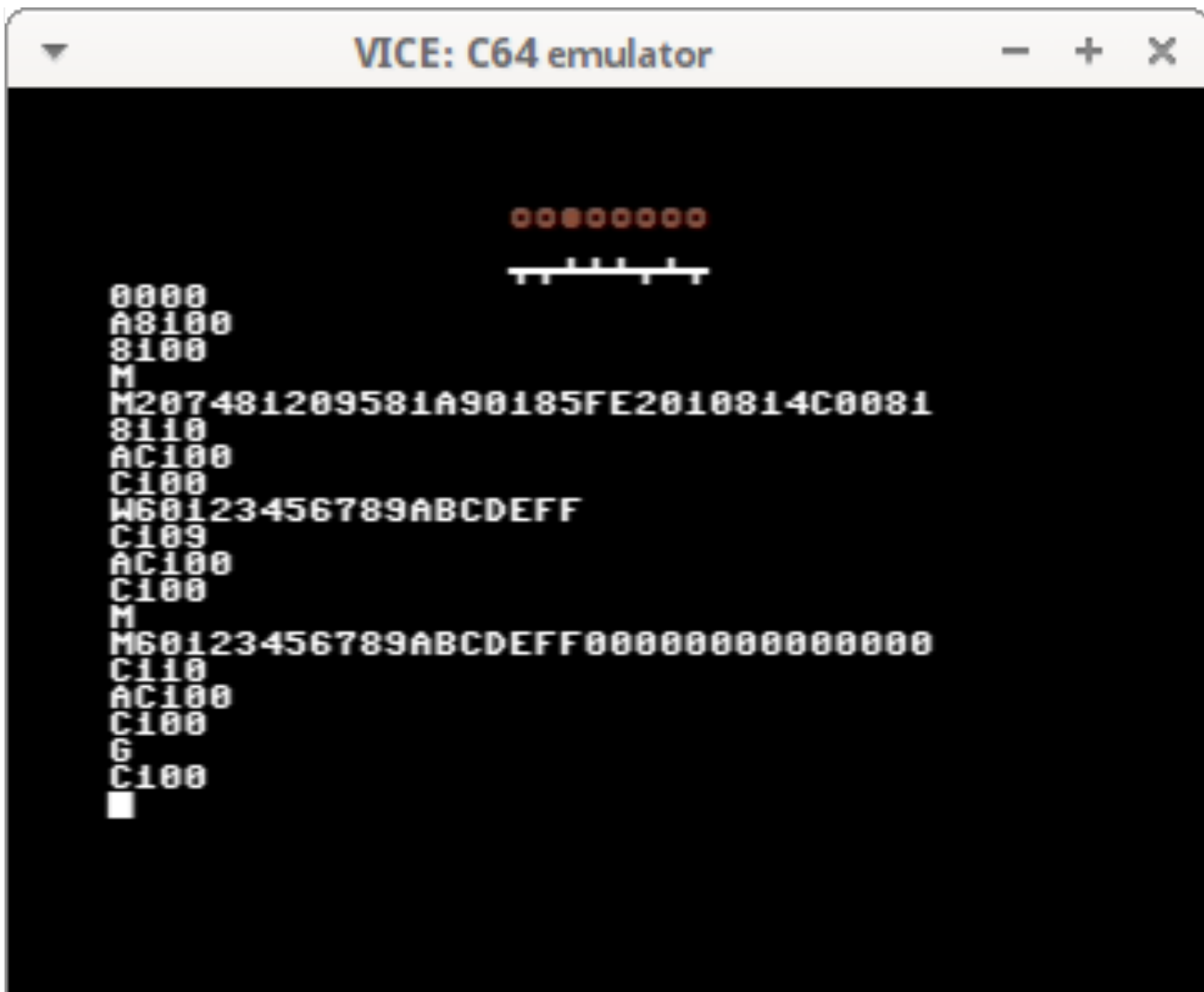
And then, back to the front-panel simulator. Reset the address to the

beginning. Keep advancing the address until you get to the next byte that needs correcting, enter the new byte, repeat. A little nerve-wracking, but not terribly difficult, because I was getting pretty used to this by now...

Then, reset the address to the beginning again, and... execute!



Hey! That looks... like what I expected! Let's see if it works:



It works! It works!

This is a very nice feeling – one that makes up for the earlier disappointments.

So now... we have a machine-code monitor! It's crude and ugly, but it does the trick, and it's small – 250 bytes, or maybe 251 bytes now, because of the addressing correction? I'm not sure, but it's still less than 255 bytes, and that's great.

ENTRY POINTS IN SITU-MON (CORRECTED)

MAIN	\$8100	
DISPATCH	\$8110	
READ_CHAR	\$8130	
READ_HEX	\$8139	
PARSE_HEX	\$813C	
READ_BYTE	\$814D	
WRITE_BYTE	\$815C	
WRITE_HEX	\$8165	
WRITE_CHAR	\$816B	
WRITE_PROMPT	\$8174	
OUTPUTLN_TTY	\$8182	
OUTPUT_TTY	\$8187	
INPUT_TTY	\$8195	
SET_ADDR	\$81A0	
READ_MEM	\$81B3	
WRITE_MEM	\$81C5	
INCR_ADDR	\$81DC	
HEXDIGITS	\$81EA	(not callable)

LINE	\$C000	
ADDR (-Lo)	\$ FB	
ADDR_HI	\$ FC	
TMP_A	\$ FD	
LINE_OFF	\$ FE	(only used in READ_BYTE)

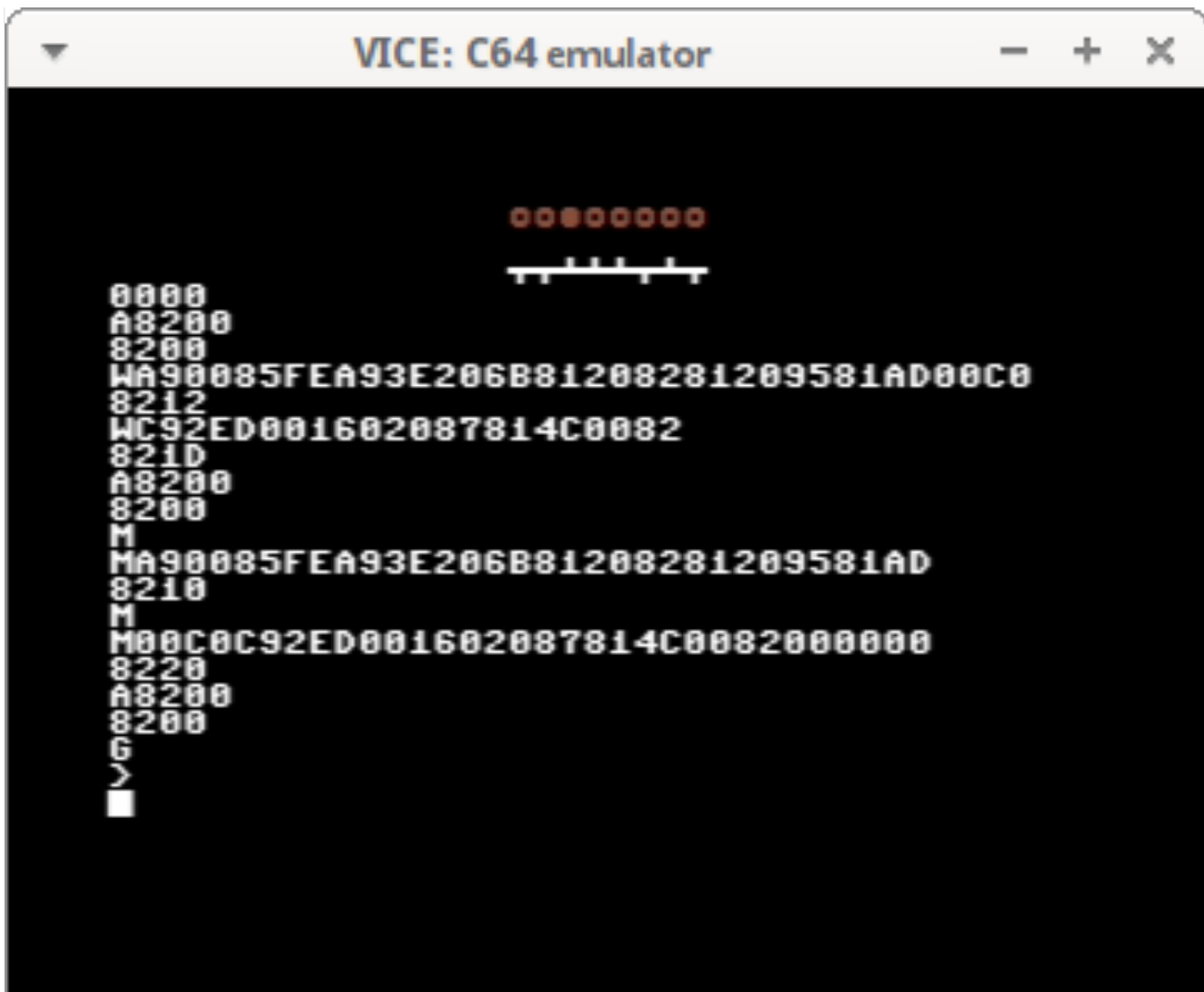
We're not done, though! A machine-language monitor is a *kind of* a programming environment, but it's not a very good one. We won't have to convert anything to binary, and entering lines on a keyboard

is a lot easier than using front-panel switches. But we do still need to hand-assemble our code to hex, and compute addresses and so forth, which means we'll still be using paper a lot.

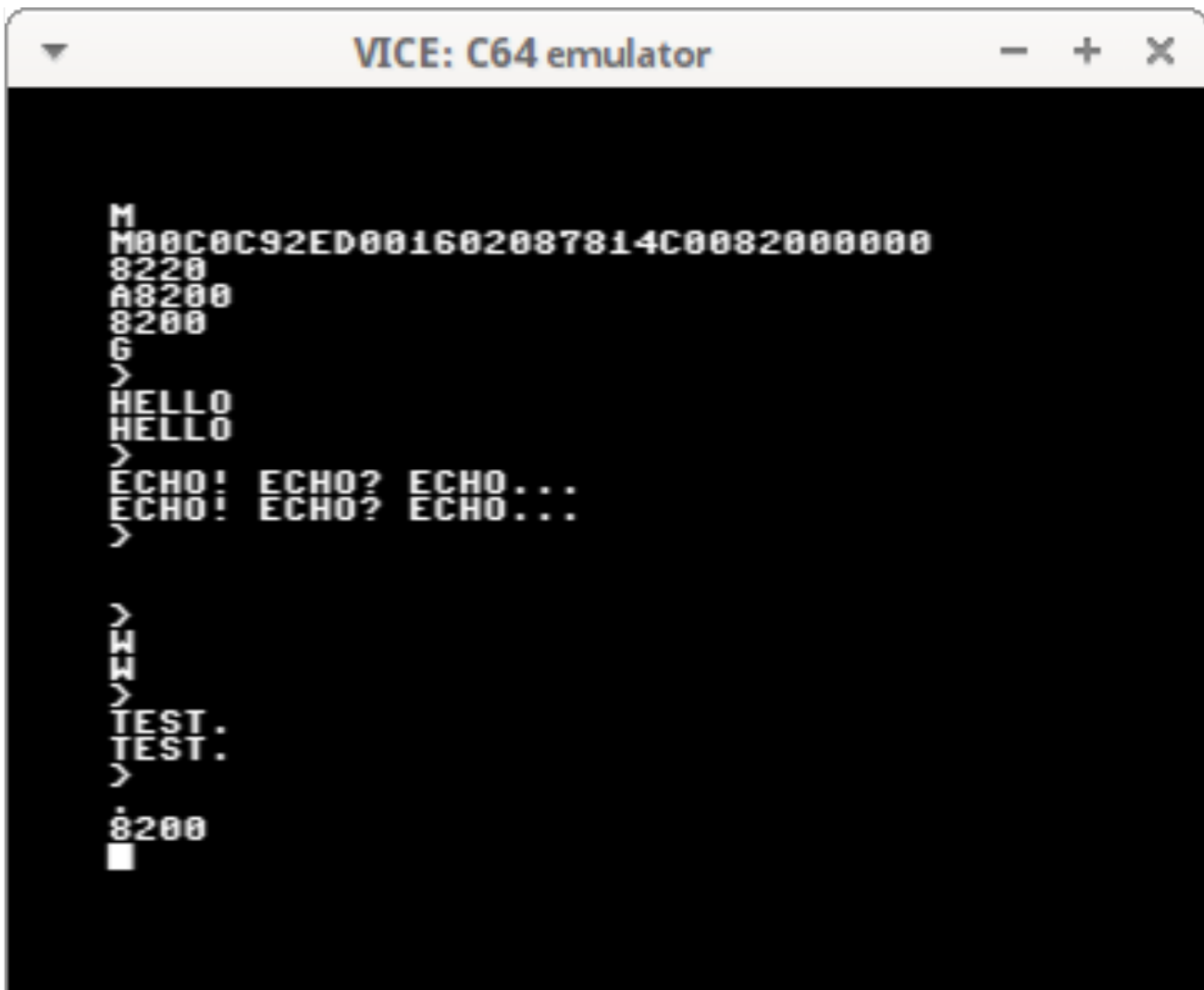
Here's an example of how much easier it will be, though. It's a fancier version of the warm-up exercise, the "echo" program. It displays a sort of prompt between lines, and lets you quit by starting the line with a period.

\$8200	ECHO2:	LDA #0	A9 00
2		STA LINE_OFF	85 FE
4		LDA #'>	A9 3E
6		JSR WRITE_CHAR	20 68 81
9		JSR OUTPUTLN_TTY	20 82 81
C		JSR INPUT_TTY	20 95 81
F		LDA LINE	AD 00 C0
8212		CMP #'.	C9 2E
4		BNE CONT	D0 01
6		RTS	60
8217	CONT:	JSR OUTPUT ~~LN~~ _TTY	20 ~~82~~ ⁸⁷ 81
821A		JMP ECHO2	4C 00 82

Note that, since we don't have to convert to binary anymore, we can format the listing on paper more compactly. Entering and reviewing and running it is a matter of a few monitor commands:



And we test it a bit to make sure that it works as designed...



I'm quite happy with this!

Now, to design and write a *better* programming environment that we can enter into the computer using SITU-MON here...

SITU-MON

OK, LET'S DESIGN & WRITE A SIMPLE
PROGRAMMING ENVIRONMENT — A
MACHINE-LANGUAGE MONITOR.

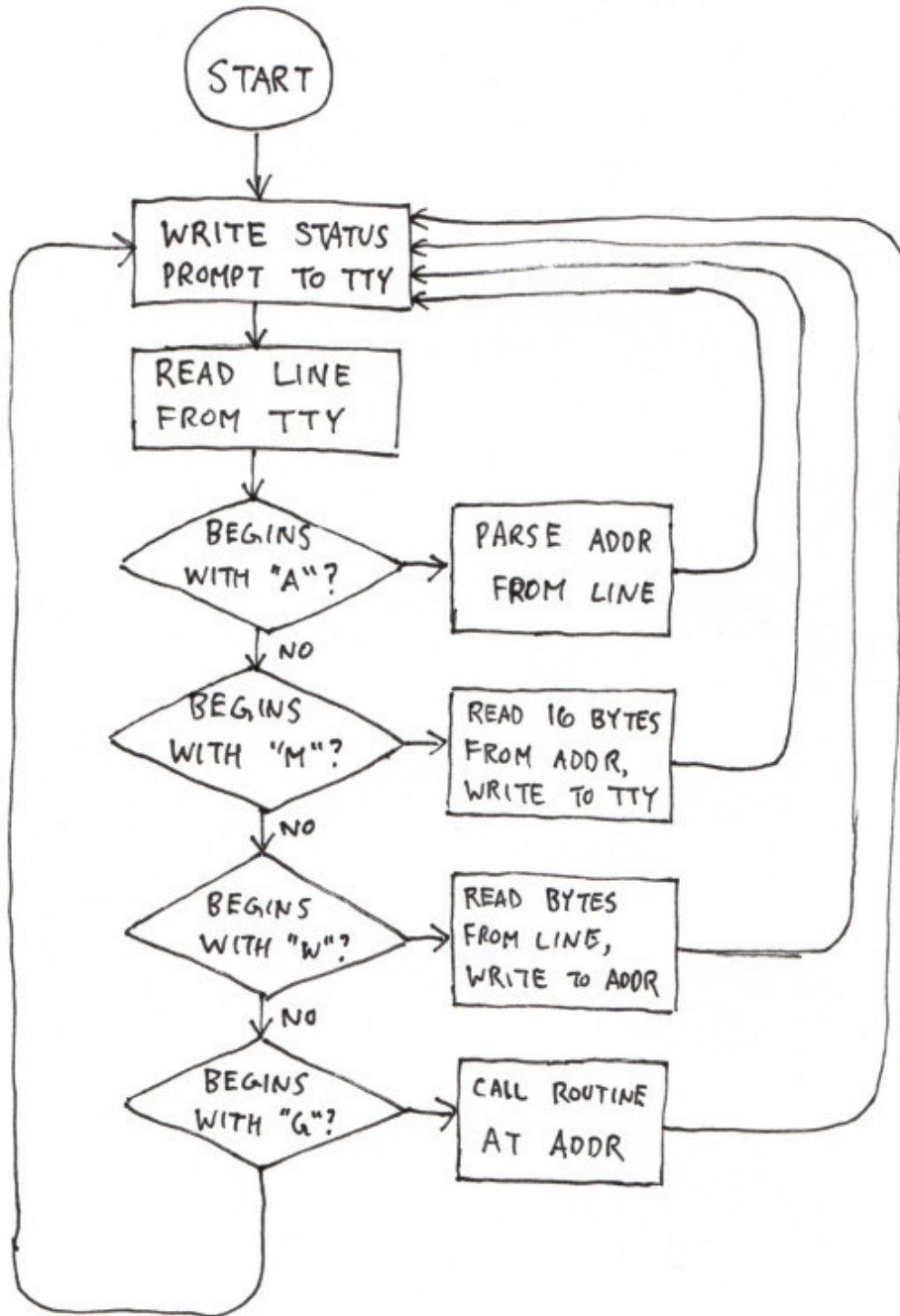
(A.K.A. DEBUGGER. NOT TO BE CONFUSED
WITH A VIDEO MONITOR, OF COURSE.)

IT WILL LACK MANY FEATURES OF A
"REAL" MONITOR, BUT IT WILL LET US:

- ENTER BYTES IN HEXADECIMAL AND
WRITE THEM INTO MEMORY
- INSPECT THE CONTENTS OF MEMORY
AS HEXADECIMAL
- CALL MACHINE-LANGUAGE ROUTINES

...AND IT WILL BE SMALL ENOUGH THAT
I'LL BE ABLE TO ENTER IT, IN BINARY,
USING FRONT-PANEL SWITCHES, WITHOUT
GOING MAD. MAYBE.

SITU - MON.



... SO, WE'LL NEED:

- A MAIN LOOP WITH CODE TO DISPATCH TO THE VARIOUS MONITOR COMMANDS (FOUNTAIN PEN)
- ROUTINES TO PARSE THE LINE THE USER ENTERED, FOR HEX DIGITS ETC. (GREEN)
- ROUTINES TO FORMAT OUTPUT FOR THE USER, AS HEX DIGITS ETC. (BLUE)
- ROUTINES TO TAKE INPUT FROM, AND SEND OUTPUT TO, THE VIRTUAL TELETYPE (PURPLE)
- THE MONITOR COMMANDS THEMSELVES (RED)
- SOME MISCELLANEOUS UTILITY ROUTINES (FUSCHIA OR WHATEVER THIS IS)

NOTE THAT THE CODE IS NOT COMMENTED (except where I've made last-minute alterations) BUT EACH ROUTINE SHOULD BE SIMPLE ENOUGH TO FOLLOW FOR ANYONE WHO KNOWS 6502 ASSEMBLER.

```

MAIN_LOOP:  JSR  WRITE_PROMPT
             JSR  INPUT_TTY
             LDA  #1
             STA  LINE_PTR          A.K.A. LINE_OFF
             JSR  DISPATCH
             JMP  MAIN_LOOP

DISPATCH:  LDA  LINE
             CMP  #'A
             BNE  NEXT
             JMP  SET_ADDR

NEXT:       CMP  #'M
             BNE  NEXT
             JMP  DUMP_MEM          A.K.A. READ_MEM

NEXT:       CMP  #'W
             BNE  NEXT
             JMP  WRITE_MEM

NEXT:       CMP  #'G
             BNE  NEXT
             JMP  (ADDR)

NEXT:       RTS

```

```
READ_CHAR:    LDY LINE_OFF
              LDA LINE, Y
              INY
              STY LINE_OFF
              RTS
```

```
READ_HEX:     JSR READ_CHAR
```

fallthrough

```
PARSE_HEX:    LDY #0
              LOOP: CMP HEXDIGITS, Y
              BNE NEXT
              TYA
              RTS
              NEXT: INY
              CPY #1
              BNE LOOP
              LDA #0
              RTS
```

```
READ_BYTE:    JSR READ_HEX
              ASL A
              ASL A
              ASL A
              ASL A
              STA TEMP_A
              JSR READ_HEX
              ORA TEMP_A
              RTS
```

WRITE_BYTE:

PHA
LSR A
LSR A
LSR A
LSR A
JSR WRITE_HEX
PLA

WRITE_HEX:

fall through
AND #\$0F
TAY
LDA HEXDIGITS, Y

WRITE_CHAR:

fall through
LDY LINE_OFF
STA LINE, Y
INY
STY LINE_OFF
RTS

```

WRITE_PROMPT:    LDA  $$00
                  STA  LINE_OFF
                  LDA  ADDR_HI
                  JSR  WRITE_BYTE
                  LDA  ADDR_LO
                  JSR  WRITE_BYTE

```

```

fallthrough
OUTPUTLN_TTY:    LDA  $$0D
                  JSR  WRITE_CHAR

```

```

fallthrough
OUTPUT_TTY:      LDY  $$00
LOOP:            LDA  LINE, Y
                  JSR  CHROUT
                  INY
                  CMP  $$0D
                  BNE  LOOP
                  RTS

```

```

INPUT_TTY:       LDY  $$00
LOOP:            JSR  CHRIN
                  STA  LINE, Y
                  INY
                  CMP  $$0D
                  BNE  LOOP
JMP  CHROUT
                  STA  LINE, Y
                  JMP  CHROUT

```

< for safety
during WRITE_MEM

```

SET_ADDR:      JSR  READ_BYTE
                STA  ADDR_HI
                JSR  READ_BYTE
                STA  ADDR_LO
                RTS

```

```

READ_MEM:      LDX  #$10
                LDY  #$00
                LDA  (ADDR), Y
                JSR  WRITE_BYTE
                JSR  INCR_ADDR
                DEX
                BNE  LOOP
                JMP  OUTPUTLN_TTY

```

```

WRITE_MEM:     LDY  LINE_OFF
                LDA  LINE, Y
                CMP  #$0D
                BNE  CONT
                RTS

```

```

CONT:          JSR  READ_BYTE
                LDY  #$00
                STA  (ADDR), Y
                JSR  INCR_ADDR
                JMP  WRITE_MEM

```


INCR_ADDR:

```
CLC
LDA ADDR_LO
ADC #$01
STA ADDR_LO
LDA ADDR_HI
ADC #$00
STA ADDR_HI
RTS
```

HEXDIGITS:

#\$30	'0
#\$31	'1
#\$32	'2
#\$33	'3
#\$34	'4
#\$35	'5
#\$36	'6
#\$37	'7
#\$38	'8
#\$39	'9
#\$41	'A
#\$42	'B
#\$43	'C
#\$44	'D
#\$45	'E
#\$46	'F

NOW, DATA....

LINE WILL BE THE SAME AS IN THE
WARM-UP: \$C000

THERE ARE ONLY 4 OTHER MEMORY LOCATIONS
THAT ARE USED, AND WE CAN PUT THEM IN
THE ZERO PAGE TO KEEP THE CODE SMALL.

ADDR HAS ITS LOW BYTE AT \$FB
AND ITS HIGH BYTE AT \$FC

TMP_A IS STORED AT..... \$FD

LINE_OFF IS STORED AT.... \$FE

WITH THESE CHOICES, THE SIZE OF THE
CODE COMES TO...

250 BYTES!

THAT'S 2,000 BITS!

THAT'S DOABLE... I THINK...

\$8100	MAIN	JSR \$8173	20 73 81	00000000	01110011	10000001
8103		JSR \$8194	20 94 81	00100000	10010100	10000001
8106		LDA #01	A9 01	10101001	00000001	
8108		STA \$FE	85 FE	10000101	11111110	
810A		JSR \$8110	20 10 81	00100000	00010000	10000001
810D		JMP \$8100	4C 00 81	01001100	00000000	10000001
8110	DISP	LDA \$C000	AD 00 C0	10101101	00000000	11000000
8113		CMP #041	C9 41	11001001	01000001	
8115		BNE \$03	D0 03	11010000	00000011	
8117		JMP \$81A7	4C A7 81	01001100	10100111	10000001
811A		CMP #04D	C9 4D	11001001	01001101	
811C		BNE \$03	D0 03	11010000	00000011	
811E		JMP \$81B2	4C B2 81	01001100	10110010	10000001
8121		CMP #057	C9 57	11001001	01010111	
8123		BNE \$03	D0 03	11010000	00000011	
8125		JMP \$81C4	4C C4 81	01001100	11000100	10000001
8128		CMP #047	C9 47	11001001	01000111	
812A		BNE \$03	D0 03	11010000	00000011	
812C		JMP (\$00FB)	6C FB 00	01101100	11111011	00000000
812F		RTS	60	01100000		
8130	RD. CH.	LDY \$FE	A4 FE	10100100	11111110	
8132		LDA \$C000, Y	B9 00 C0	10111001	00000000	11000000
8135		INY	C8	11001000		
8136		STY \$FE	84 FE	10000100	11111110	
8138		RTS	60	01100000		
8139	RD. HX.	JSR \$8130	20 30 81	00100000	00110000	10000001
813B	PS. HX.	LDY #000	A0 00	10100000	00000000	
813D		CMP \$81E9, Y	D9 E9 81	11011001	11101001	10000001
8140		BNE \$02	D0 02	11010000	00000010	

\$8142		TYA	98	10011000		
8143		RTS	60	01100000		
8144		INY	C8	11001000		
8145		CPY #010	C0 10	11000000	00010000	
8147		BNE \$F4	D0 F4	11010000	11110100	
8149		LDA #000	A9 00	10101001	00000000	
814B		RTS	60	01100000		
814C	RD. BY.	JSR \$8139	20 39 81	00100000	00111001	10000001
814F		ASL A	0A	00001010		
8150		ASL A	0A	00001010		
8151		ASL A	0A	00001010		
8152		ASL A	0A	00001010		
8153		STA \$FD	85 FD	10000101	11111101	
8155		JSR \$8139	20 39 81	00100000	00111001	10000001
8158		ORA \$FD	05 FD	00000101	11111101	
815A		RTS	60	01100000		
815B	WR. BY.	PHA	48	01001000		
815C		LSR A	4A	~~01000000~~	01001010	
815D		LSR A	4A	~~01000000~~	01001010	
815E		LSR A	4A	~~01000000~~	01001010	
815F		LSR A	4A	~~01000000~~	01001010	
8160		JSR \$8164	20 64 81	00100000	01100100	10000001
8163		PLA	68	01101000		
8164	WR. HX.	AND #00F	29 0F	00101001	00001111	
8166		TAY	A8	10101000		
8167		LDA \$81E9, Y	B9 E9 81	10111001	11101001	10000001
816A	WR. CH.	LDY \$FE	A4 FE	10100100	11111110	
816C		STA \$C000, Y	99 00 C0	10011001	00000000	11000000
816F		INY	C8	11001000		
8170		STY \$FE	84 FE	10000100	11111110	
8172		RTS	60	01100000		

\$ 8173	WR.PR.	LDA #00	A9 00	10101001	00000000	
8175		STA \$FE	85 FE	10000101	11111110	
8177		LDA \$FC	A5 FC	10100101	11111100	
8179		JSR \$815B	20 5B 81	00100000	01011011	10000001
817C		LDA \$FB	A5 FB	10100101	11111011	
817E		JSR \$815B	20 5B 81	00100000	01011011	10000001
8181	O.PLN	LDA #00	A9 00	10101001	00001101	
8183		JSR \$816A	20 6A 81	00100000	01101010	10000001
8186	O.P.TTY	LDY #00	A0 00	10100000	00000000	
8188		LDA \$C000,Y	B9 00 C0	10111001	00000000	11000000
818B		JSR \$FFD2	20 D2 FF	00100000	11010010	11111111
818E		INY	C8	11001000		
818F		CMP #00	C9 00	11001001	00001101	
8191		BNE \$F5	D0 F5	11010000	11110101	
8193		RTS	60	01100000		
8194	IMP.TTY	LDY #00	A0 00	10100000	00000000	
8196		JSR \$FFCF	20 CF FF	00100000	11001111	11111111
8199		STA \$C000,Y	99 00 C0	10011001	00000000	11000000
819C		INY	C8	11001000		
819D		CMP #00	C9 00	11001001	00001101	
819F		BNE \$F5	D0 F5	11010000	11110101	
81A1		STA \$C000,Y	99 00 C0	10011001	00000000	11000000
81A4		JMP \$FFD2	4C D2 FF	01001100	11010010	11111111
81A7	SET AD.	JSR \$814C	20 4C 81	00100000	01001100	10000001
81AA		STA \$FC	85 FC	10000101	11111100	
81AC		JSR \$814C	20 4C 81	00100000	01001100	10000001
81AF		STA \$FB	85 FB	10000101	11111011	
81B1		RTS	60	01100000		
81B2	RD. MEM	LDX #01	A2 10	10100010	00010000	
81B4		LDY #00	A0 00	10100000	00000000	

\$ 81B6		LDA (\$FB), Y	B1 FB	10110001	11111011	
81B8		JSR \$815B	20 5B 81	00100000	01011011	10000001
81BB		JSR \$81DB	20 DB 81	00100000	11011011	10000001
81BE		DEX	CA	11001010		
81BF		BNE \$F3	D0 F3	11010000	11110011	
81C1		JMP \$8181	4C 81 81	01001100	10000001	10000001
81C4	WR.MEM	LDY \$FE	A4 FE	10100100	11111110	
81C6		LDA \$C000,Y	B9 00 C0	10111001	00000000	11000000
81C9		CMP #00	C9 00	11001001	00001101	
81CB		BNE \$01	D0 01	11010000	00000001	
81CD		RTS	60	01100000		
81CE		JSR \$814C	20 4C 81	00100000	01001100	10000001
81D1		LDY #00	A0 00	10100000	00000000	
81D3		STA (\$FB), Y	91 FB	10010001	11111011	
81D5		JSR \$81DB	20 DB 81	00100000	11011011	10000001
81D8		JMP \$81C4	4C C4 81	01001100	11000100	10000001
81DB	INCR	CLC	18	00011000		
81DC		LDA \$FB	A5 FB	10100101	11111011	
81DE		ADC #01	69 01	01101001	00000001	
81E0		STA \$FB	85 FB	10000101	11111011	
81E2		LDA \$FC	A5 FC	10100101	11111100	
81E4		ADC #00	69 00	01101001	00000000	
81E6		STA \$FC	85 FC	10000101	11111100	
81E8		RTS	60	01100000		
81E9	HEXDIG.	\$30	30	00110000	39	00111001
		\$31	31	00110001	41	01000001
		\$32	32	00110010	42	01000010
		\$33	33	00110011	43	01000011
		\$34	34	00110100	44	01000100
		\$35	35	00110101	45	01000101
		\$36	36	00110110	46	01000110
		\$37	37	00110111		
		\$38	38	00111000		

and now,
to enter it ...

Cheatsheets and Chickenscratches

I decided, while I was writing all this code by hand, I might as well also write out some stuff about 6502 programming that I can never remember when I need it.

6502 CONDITIONAL BRANCHES

THE OPERAND OF A CONDITIONAL BRANCH IS A RELATIVE ADDRESS, GIVEN AS A SIGNED BYTE ENCODED IN TWO'S COMPLEMENT. THE ADDRESS IS RELATIVE TO THE INSTRUCTION FOLLOWING THE BRANCH. THUS, -2 = \$FE WILL BRANCH TO ITSELF.

-1	FF	-24	E8
-2	FE	-25	E7
-3	FD	-26	E6
-4	FC	-27	E5
-5	FB	-28	E4
-6	FA	-29	E3
-7	F9	-30	E2
-8	F8	-31	E1
-9	F7	-32	E0
-10	F6	-33	DF
-11	F5	-34	DE
-12	F4	-35	DD
-13	F3	-36	DC
-14	F2	-37	DB
-15	F1	-38	DA
-16	F0	-39	D9
-17	EF	-40	D8
-18	EE	-41	D7
-19	ED	-42	D6
-20	EC	-43	D5
-21	EB	-44	D4
-22	EA	-45	D3
-23	E9	-46	D2

WHEN COMPARING SIGNED BYTES, CHECK THE
CARRY FLAG TO TEST FOR RELATIVE MAGNITUDE.

```
LDA    FOO
CMP    #$06
BCC    FOO_LT_6
:
```

```
LDA    BAR
CMP    #$06
BCS    BAR_GTE_6
:
```

```
LDA    BAZ
CMP    #$06
BEQ    BAZ_EQ_6
BCS    BAZ_GT_6
:
```

TWO TECHNIQUES FOR SHORTENING MACHINE CODE.

1. FALLTHROUGH

FOO: BLAH
BLAH
JMP BAR

BAR: BLAH
BLAH
:



FOO: BLAH
BLAH

BAR: BLAH
BLAH
:

2. TAIL CALL

FOO: BLAH
BLAH
JSR BAR
RTS

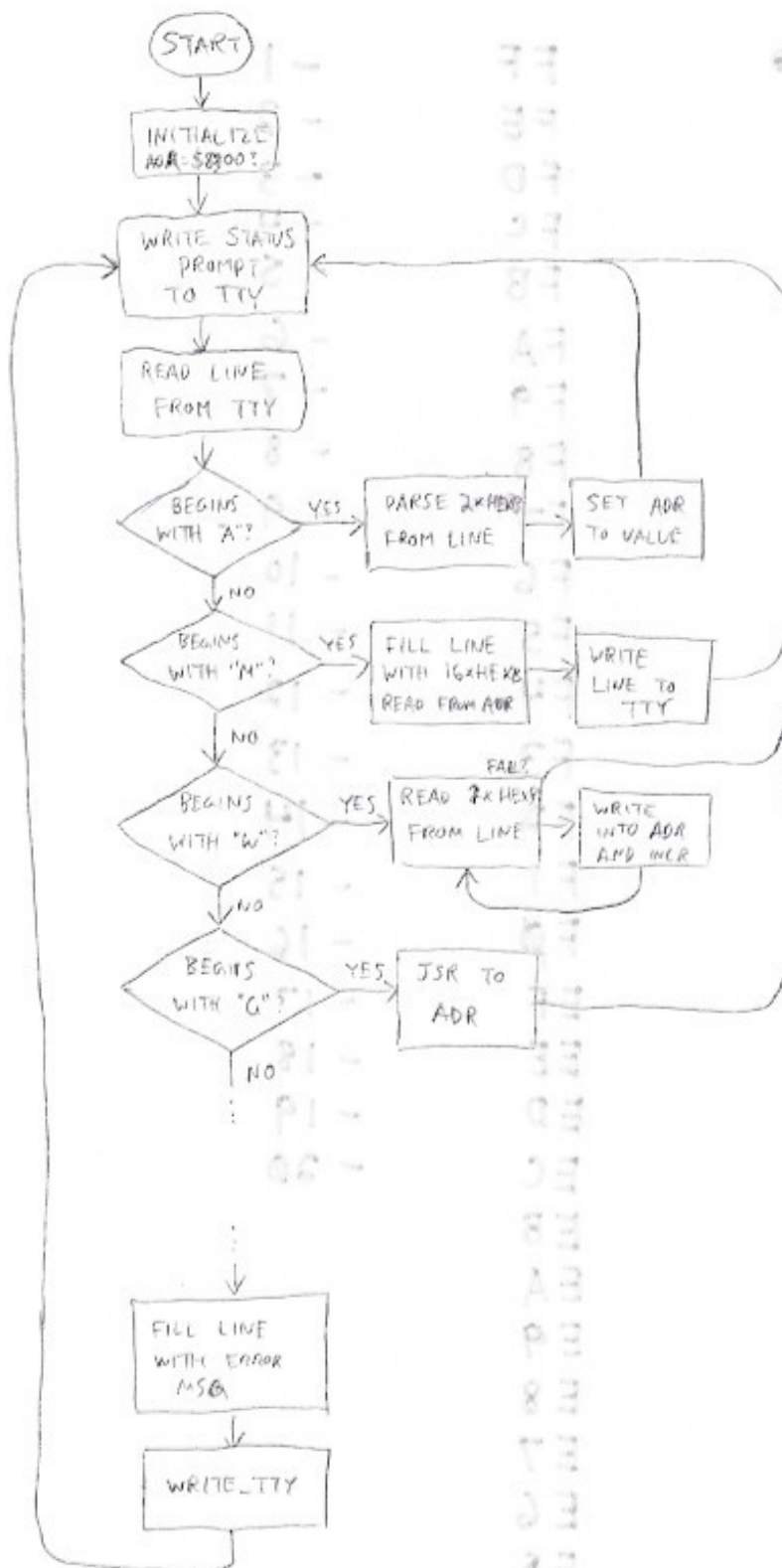


FOO: BLAH
BLAH
JMP BAR

EASY TO REMEMBER BUT EVEN EASIER TO LOOK UP....

Ø	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Sketches of the design and code for the machine-language monitor.
It was basically written three times; once in pencil, once with a ballpoint pen, and the final version (not shown here, coming soon!) with fineliners.



WRITE_NBBLE:

AND #\$0F

TAY

LDA HEXDIGITS, Y

~~LDY LINE_PTR~~

~~STA LINE, Y~~

~~INY~~

~~STY LINE_PTR~~

~~RTS~~

JMP WRITE_CHAR

WRITE_CHAR

LDY LINE_PTR

~~LDA #CR~~

STA LINE, Y

INY

STY LINE_PTR

RTS

WRITE_EOL

LDA #\$0D

JMP WRITE_CHAR

WRITE_BYTE_X

TXA

JSR

LSR

LSR

LSR

JSR WRITE_NYB

TXA

JSR WRITE_NYB

RTS

WRITE_PROMPT

LDX ADDR_HD

JSR WRITE_BYTE_X

LDX ADDR_LO

JSR WRITE_BYTE_X

RDA #":"

JSR WRITE_CHAR

~~RTS~~

JMP WRITE_EOL

DISPATCH

JSR WRITE_PROMPT

JSR READ_TTY

LDA LINE

CMP # "A"

BNE NEXT

~~JMP~~ SET_ADDR

NEXT: CMP # "M"

BNE NEXT

JMP

:

JSR WRITE_ERR

JMP DISPATCH

READ-CHAR:

```
LDY LINE_PTR
LDA LINE, Y
INY
STY LINE_PTR
RTS
```

PARSE_HEX: (output in Y)

```
LDY #0
LOOP: CMP HEXDIGITS, Y
      BNE NEXT
      RTS ← NEXT
NEXT: INY ← BCC
      CLC
      BCC LOOP
```

```
EXIT: T
      CPY #$10
      BNE LOOP
      RTS ← SEC
```

READ_BYTE:

```
JSR READ_CHAR
JSR PARSE_HEX; TYA BCC CONT
CONT: RTS ASL TYA; ASL
      ASL
      ASL
      SAA TEMP
      JSR READ_CHAR
      JSR PARSE_HEX; TYA BCC CONT
      RTS
CONT: ORA TEMP; SEC
      RTS
```

SET-ADR:

```
LDA #0
STA LINE_PTR
JSR READ_BYTE BCS ERR
STA ADR_HI
JSR READ_BYTE BCS ERR
STA ADR_LO
```

ERR: RTS

GO:

```
JMP (ADR_LO)
RTS
```

WRITE_MEM:

```
JSR READ_BYTE
BCC CONT
CONT: RTS STA (ADR_LO)
      CLC
      LDA ADR_LO
      ADC #1
      STA ADR_LO
      LDA ADR_HI
      ADC #0
      STA ADR_HI
      JMP WRITE_MEM
```

READ_MEM:

```
LDX #0
LOOP: LDA (ADR_LO) TAX
      JSR WRITE_BYTE_X
      JSR INC_ADR
      DEY
      BNE LOOP
      JSR WRITE_NL
      JSR WRITE_TTY
      RTS
```



```

READ_CHAR:    LDY LINE_OFF      ; output: A = char
              LDA LINE, Y       ; increases line offset
              INY
              STY LINE_OFF
              RTS

```

```

PARSE
READ HEX:    LDY #0           ; input: A = char
              LOOP:  CMP HEXDIGITS, Y   ; output: A = hex value
                   BN  LDY #0
                   CMP HEXDIGITS, Y
                   BEG FOUND
                   DEY

```

```

PARSE_HEX:    LDY #0
              LOOP:  CMP HEXDIGITS, Y
                   BNE NEXT
                   TYA
                   RTS
              NEXT:  INY
                   CPY #10
                   BNE LOOP
                   <LDA #0?>
                   RTS

```

M.L.	3	92
	16	24
DISP.	32	49
R.C.	9	
R.H.	3	~~165~~
P.H.	17	~~16~~
R.B.	15	~~181~~
		<u>92</u>
W.B.	9	
W.H.	6	
W.C.	9	
		<u>24</u>
W.PR.	14	
OUT LN	5	
OUT TT	14	
INTT	16	
		<u>49</u>

	READ_BYTE:	~~JSR READ_CHAR~~	SET_ADDR	11
		~~JSR PARSE_HEX~~	RD	19
		~~ASL~~	WR	23
		~~ASL~~		<u>52</u>
		~~ASL~~	INCR	14
		~~ASL~~	HEX	16
		STA TEMP_A		<u>30</u>
		JSR READ_CHAR		
		JSR PARSE_HEX		

			24
READ_BYTE:	JSR READ_CHAR		49
	JMP PARSE_HEX		52
			<u>30</u>
READ_BYTE:	JSR READ_HEX		247
	ASL (x4)		250

	STA TEMP_A
	JSR READ_HEX
	ORA TEMP_A
	RTS
SET_ADDR:	JSR READ_BYTE
	STA ADDR_HI
	JSR READ_BYTE
	STA ADDR_LO
	RTS

WRITE_BYTE: ~~STA TEMP-A~~ PHA
WRITE_NYBBLE JSR
~~LDA TEMP-A~~ PLA
LSR
LSR
LSR
LSR
JMP WRITE_NYBBLE

PHA
LSR x4
JSR WRITE_NYB
PLA
JMP WRITE_NYB

WRITE_NYB: AND #0F
TAY
LDA HEXDIGITS, Y
JMP WRITE_CHAR

WRITE_CHAR: LDY LINE_OFF
STA LINE, Y
INY
STY LINE_OFF
RTS

OUTPUTLN_TTY: LDA #CR
JSR WRITE_CHAR
JMP OUTPUT_TTY

```

WRITE_MEM:    LDY LINE_OFF
               LDA LINE, Y
               CMP #CR
               BNE CONT
               RTS
CONT:         JSR READ_BYTE ← LDY #0
               STA (ADDR), Y
               JSR INCR_ADDR
               JMP WRITE_MEM

```

```

INCR_ADDR:    CLC
               LDA ADDR_LO
               ADC #01
               STA ADDR_LO
               LDA ADDR_HI
               ADC #00
               STA ADDR_HI
               RTS

```

```

READ_MEM:    LDX #0 ← LDY #0
Loop: → LDA (ADDR), Y
           JSR WRITE_BYTE
           DEX JSR INCR_ADDR
           BNE DEX
           BNE Loop
           JMP OUTPUT_TTY OUTPUTLN_TTY

```

```

                LDA #0
                STA LINE_OFF
WRITE_PROMPT:   LDA ADDR_HI
                JSR WRITE_BYTE
                LDA ADDR_LO
                JSR WRITE_BYTE
                JMP OUTPUTLN_TTY

```

MEMORY

\$C000

LINE

\$FB	ADDR, ADDR_LO
\$FC	ADDR_HI
\$FD	TMP_A
\$FE	LINE_OFF

77	01001101
$\frac{64}{13}$	\$4D
$\frac{8}{5}$	
	87 0100111
	$\frac{64}{23}$ \$57
	$\frac{16}{7}$

A better front-panel simulator

One thing that disappointed me slightly during [The Warm-up](#) was that the simulated front-panel loader was not terribly realistic. It lets you backspace to erase bits you've typed in, and it leaves a "paper trail" of values you've entered, on the screen. It occurred to me that it wouldn't be too difficult to write something that simulates a front panel better, so that is just what I did.

The front-panel simulator below works a lot like the 1802 Membership Card front panel (see [Loading the code](#) for more details on that.)

```
10 S=33024:A=S:D=0:TB$=""
15 UP$="SDFGHJKL":DN$="ZXCUBNM,"
20 POKE53281,0:POKE53280,0:PRINT"
30 U=PEEK(A):P=128:LS="":DS=""
40 FORI=1TO8
50 QS="0":IF(UANDP)=PTHENQS="●"
60 SS="1":IF(DANDP)=PTHENS$="T"
70 LS=LS+QS:DS=DS+SS:P=P/2
80 NEXT:PRINT"ST"TB$:TB$:LS
85 PRINT"U":TB$:TB$:DS
90 GET K$:IFK$=""GOTO 90
100 P=128:OD=D
110 FORI=1TO8
120 IFK$=MID$(DN$,I,1)THEND=DORP
130 IFK$=MID$(UP$,I,1)THEND=DAND(255-P)
140 P=P/2:NEXT
150 IFD<>ODGOTO30
160 IFK$=CHR$(13)THENPOKEA,D
170 IFK$="="THENA=A+1
180 IFK$="&"THENA=S
185 IFK$="!"THENSYSA:END
190 GOTO30
READY.
```

The screen shows an approximation of 8 LEDs and 8 SPST switches that would be connected to the 8 data lines on a real front panel.



The keys on the bottom row of the keyboard move each switch down; the keys in the 2nd-to-bottom row move each switch up. (In an effort to be more realistic, they are not toggled by a single key.)



Pressing Return causes the binary value selected by the switches to be written to the current address in RAM. That value is then reflected in the LEDs.



Other simulated buttons (not show in the UI) are

- `=` , which advances the address by one (you typically press this every time you are happy with the current byte you have written,)
- `&` which resets the address to the beginning of memory (which can be used to review the bits you've entered by stepping through them with `=` and watching the LEDs), and
- `!` to execute the machine code at the current address.

I plan to use it to enter the machine-code monitor program I'm writing (more on that in a future post.)

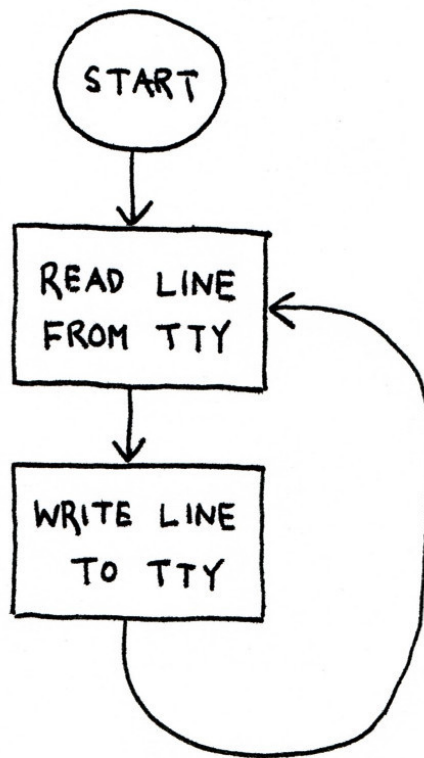
I'll put the code for this front-panel simulator (and other code from this project) in a repository on Github at the end of the month (or earlier, if someone requests it.)

The Warm-up

BEFORE EMBARKING ON OUR MAIN PROJECT,
WE'RE GOING TO DO A LITTLE WARM-UP:

- DESIGN AN "ECHO" PROGRAM
FOR THE SIMULATED TELETYPE
- CODE IT, BY HAND, IN 6502 ASSEMBLY
- HAND-ASSEMBLE IT TO MACHINE CODE
(IN HEXADECIMAL)
- HAND-CONVERT THE MACHINE CODE
TO BINARY
- HAND-ENTER THE BINARY CODE
USING
SIMULATED
FRONT-PANEL
SWITCHES

WARM-UP.



```
START:  JSR READ_TTY
        JSR WRITE_TTY
        JMP START
```

```

READ_TTY:    LDY  #$00
              LOOP: JSR  CHRIN          $FFCF
                  STA  LINE, Y
                  INY
                  CMP  #$0D            carriage return
                  BNE  LOOP
                  RTS

```

```

WRITE_TTY:   LDY  #$00
              LOOP: LDA  LINE, Y
                  JSR  CHROUT          $FFD2
                  INY
                  CMP  #$0D
                  BNE  LOOP
                  RTS

```

```

LINE:        (256 bytes of RAM)      $C000

```


\$8100	JSR	\$8109	20 09 81	00100000	00001001	10000001
8103	JSR	\$8117	20 17 81	00100000	00010111	10000001
8106	JMP	\$8100	4C 00 81	01001100	00000000	10000001
8109	LDY	#0	A0 00	10100000	00000000	
810B	JSR	\$FFCF	20 CF FF	00100000	11001111	11111111
810E	STA	\$C000,Y	99 00 C0	10011001	00000000	11000000
8111	INY		C8	11001000		
8112	CMP	#\$0D	C9 0D	11001001	00001101	
8114	BNE	\$F5	D0 F5	11010000	11110101	
8116	RTS		60	01100000		
8117	LDY	#0	A0 00	10100000	00000000	
8119	LDA	\$C000,Y	B9 00 C0	10111001	00000000	11000000
811C	JSR	\$FFD2	20 D2 FF	00100000	11010010	11111111
811F	INY		C8	11001000		
8120	CMP	#\$0D	C9 0D	11001001	00001101	
8122	BNE	\$F5	D0 F5	11010000	11110101	
8124	RTS		60	01100000		

All written! Now, to load it (see "[Loading the code](#)" for details on this)...

```

LIST
10 S=33024:A=S
20 INPUT G$
30 IF G$="X" THEN SYS S:END
35 U=0:P=128
40 FOR I=1 TO 8
45 IF MID$(G$,I,1)="1" THEN U=U OR P
50 P=P/2
55 NEXT I
60 POKE A,U
70 A=A+1
80 GOTO 20
READY.

RUN
? 01100000
? X
READY.

```

```
RUN
001000000
00001001
100000001
001000000
00010111
100000001
01001100
000000000
100000001
101000000
000000000
001000000
11001111
11111111
10011001
000000000
110000000
11001000
11001001
00001101
11010000
11110101
```

```
11110101
011000000
101000000
000000000
10111001
000000000
110000000
001000000
11010010
11111111
11001000
11001001
00001101
11010000
11110101
011000000
X
HELLO HELLO
ECHO ECHO
WELL, IT WORKS! WELL, IT WORKS!

YES YES
IT IT
DOES DOES
```

SUCCESS!

"ECHO" PROGRAM IS 37 BYTES.

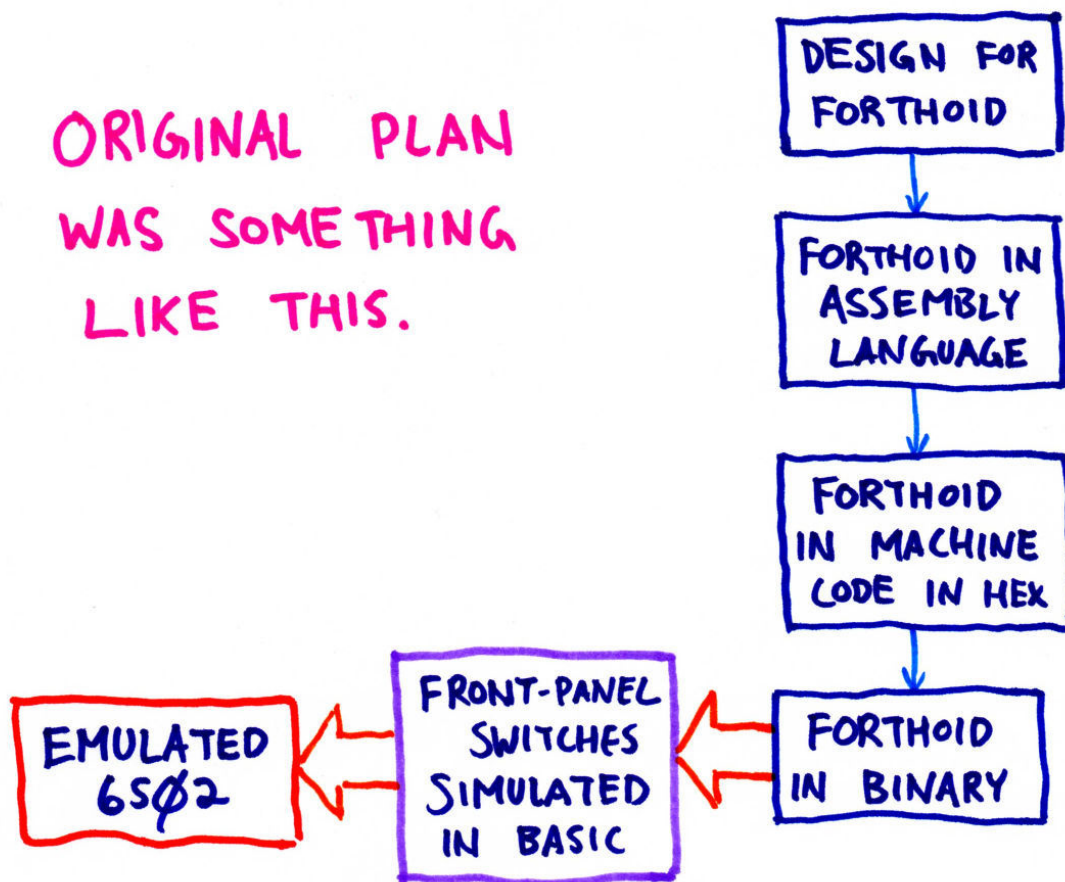
THAT'S 296 BITS.

I NEARLY MADE 2 MISTAKES, BUT
CAUGHT THEM.

THE MAIN PROJECT WILL BE MUCH
LONGER THAN THIS, THOUGH.

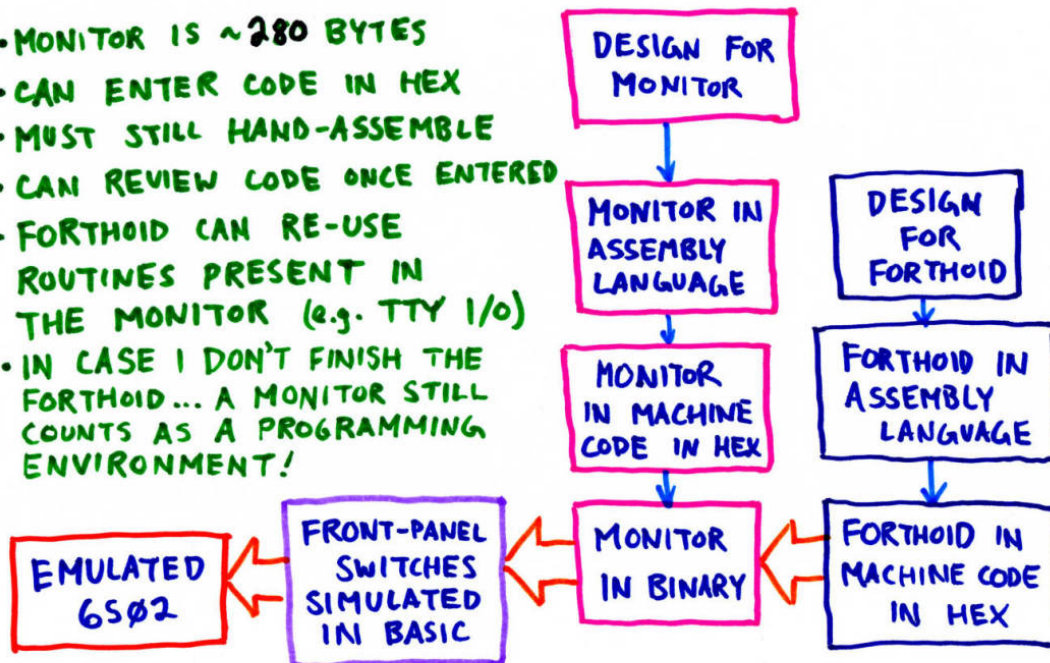
WHICH PROMPTS ME TO RE-THINK
THINGS A BIT...

ORIGINAL PLAN
WAS SOMETHING
LIKE THIS.



REVISED PLAN IS SOMETHING LIKE THIS.

- MONITOR IS ~280 BYTES
- CAN ENTER CODE IN HEX
- MUST STILL HAND-ASSEMBLE
- CAN REVIEW CODE ONCE ENTERED
- FORTHOID CAN RE-USE ROUTINES PRESENT IN THE MONITOR (e.g. TTY I/O)
- IN CASE I DON'T FINISH THE FORTHOID... A MONITOR STILL COUNTS AS A PROGRAMMING ENVIRONMENT!



Loading the code

The most “cheating” part of this project will be avoiding the actual loading of the machine code into an actual machine, so I feel that should describe what it would *actually* be like, in the real world.

For the sake of example, assume we’ve already written a programming environment — the one that we want to put into our computer, a computer that doesn’t have any sort of programming environment on it yet. Assume we’ve written this program by hand, on paper... and that we’ve already converted it all into machine code. The first line of it might look something like this:

<u>ADDRESS</u>	<u>INSTRUCTION</u>	<u>HEX</u>	<u>BINARY</u>
\$0800	LDA #\$00	A9 00	10101001 00000000

...followed by many, many more lines.

There are, of course, many ways to get a program into a computer – that is, into its memory.

In the modern world, we would just open a text editor and start typing in the program. But think of how much hardware and software is required to support this! Decoding the keystrokes from the keyboard is not terribly complicated, but displaying characters on a video display is quite another matter. And never mind that the text editor is yet another program that someone had to write (and get into the computer somehow.)

I'll describe a much more primitive method here.

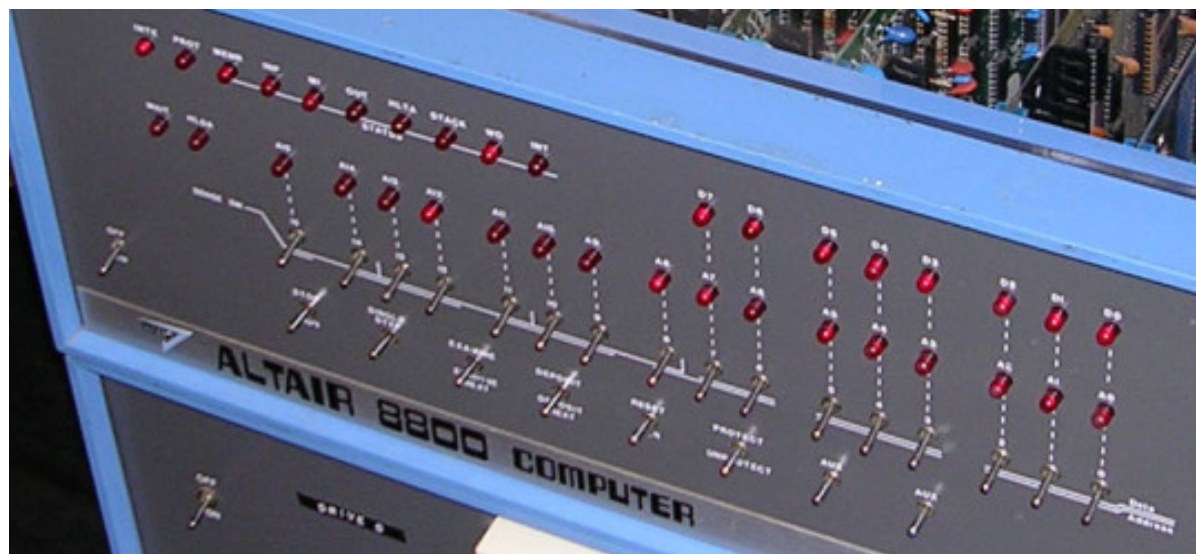
Glossing over some of the details, a typical memory circuit has a number of **address lines**, a number of **data lines**, and a **read/write gate**. For example, a 64 kilobyte bank of memory might have 16 address lines and 8 data lines.

To locate a byte in memory, you put a binary-encoded version of its address on the address lines. To read that byte, you set the read/write gate to “read” and look at what value is on the data lines. To write a byte, you instead put the value you write onto the data lines, and set the read/write gate to “write”.

Normally, the CPU is doing all this memory-access. But all these lines, **they're just wires with voltages on them**. So you can do all this reading and writing **by hand, using switches**.

In fact, the front panels of some of the earliest home computers had

switches and lights on them for just this purpose, like the Altair 8800 shown here:



And in the absence of any other hardware, **this was how you programmed it**. You flipped the switches to represent the binary value you wanted to write into memory, and pressed a button to write it.

I was going to walk through this process and ask you to imagine it, but [@retrochallenge](#) pointed out the [1802 Membership Card](#), which is a kit based on the COSMAC ELF, which, like the Altair 8800, was programmed with front panel switches.

So instead, you can [read the instructions for 1802 Membership Card code loading](#), or watch this video to see what loading code this way is like:

And now that I've shown you this, there is something I'd like to point out...

There's a simple improvement over switches that's fairly obvious when you think about it a bit. Instead of manually changing the switches with your fingers, why not have some kind of mechanical contrivance which sets the switches according to a pre-arranged pattern?

Indeed, this technology exists, and you have probably heard of it: the *punched card*.

FORTRAN STATEMENT										IDENTIFICATION									
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

At its simplest, a punchcard reader is barely any more complex than front-panel switches. When a card is inserted, switch contacts close where there are holes in the card, and stay open where there aren't holes. And then that bit pattern gets written to memory. As a bonus, the punched card serves as a record of the bytes that were written to memory, and can be re-used.

If anything, it's the technology of card *punching* that's more advanced. In the example above, the card puncher has printed at the top, by what looks like a dot-matrix printer, the characters for the values it has punched holes for below. But that's certainly not necessary; in principle, you could punch out the holes by hand, using a hole-punch.

Simulating loading the code

OK! Now that I've imparted to you what it's like, I'm going to explain why I'm going to take the liberty of not actually doing it this way, and how I plan to fudge it :)

This project is primarily about *making* a new software development environment (using the existing "software development environment" of pen and paper) to be used on a computer. The act of actually

putting it there is secondary — still important for context, of course, which is why I described it in detail.

Also, if all the computer has is front-panel switches and lights, then maybe putting a programming environment onto it isn't the most useful thing you can do. Yes, it's easier to write a program in BASIC instead of in machine language, but if you still have to enter your BASIC program, byte by byte, in binary, using switches or by punching holes, that's still quite a bottleneck.

My point here is that front-panel switches and punchcard readers can load code into a computer *directly*, at the hardware level. Most other input devices — keyboards, serial ports, etc. — can't do that; they're too general-purpose. To load code, they would need to work in concert with some software routines that interpret the input they receive.

Ah, but a programming environment could contain such software routines! In fact, if the purpose of a programming environment is to make it easier to develop software, then arguably it should let you use these easier-to-use, general-purpose input devices for this task.

So, since the hardware for this project is going to be emulated anyway, what I would like to do is assume that the machine has, along with **simulated front-panel switches**, some kind of **simulated teletype** attached to it.

We'll enter the programming environment into the computer by the front-panel switches. But the programming environment itself will use the simulated teletype to interact with the user.

The actual architecture will probably look something like this:

- An emulator for a 6052-based computer (probably a Commodore 64).

- A little program that simulates front-panel switches, reading in strings of binary digits like **00000100** from the C64 keyboard and writing values (in this case, 4) to successive locations in memory.
- A pair of machine-language subroutines for reading a line of text from, and writing a line of text to, the simulated teletype. **In principle**, we could write these subroutines so that they actually talked to a teletype. **In practice**, they'll probably be implemented with calls to C64 KERNAL routines for reading from the keyboard and writing to the screen.

If that seems rather “cheaty”, well... I agree, it's not ideal, but I've only got a month to finish this :)

Since this might be the last blog post before I begin, I'll mention a few other things you can expect from this project.

- The program will be designed, programmed, and hand-assembled, on paper.
- Updates to this blog will consist mainly of photos or scans of those papers... I won't be writing posts about the design.
- I will have a hardcopy of the Commodore 64 Programmer's Reference Guide which I will feel free to consult, but I'll try very very hard not to consult the Internet.
- I might do a little warmup first (like, “echo the lines you type in”) before starting on the programming environment proper.
- The programming environment will probably be very similar to FORTH. I will definitely be trying to keep it small (I will have to enter all these bytes, in binary, at the end!) while aiming to make it easier to program with than assembly.
- The programming environment won't use any features of the C64 except for those two KERNAL routines to simulate a teletype.

- I won't hesitate to save the emulator's state while entering the code, so I don't have to do it all in one sitting. But... I'll try very hard not to abuse that feature, to e.g. "undo" entering wrong code. (This suggests the programming environment should contain some kind of debugger...)

And, er, that's it! Can't wait to start!

Why write it by hand?

In the [previous post](#), I outlined the project that [Cat's Eye Technologies](#) plans to do for [RetroChallenge](#) 2015/07: writing a programming environment by hand.

But I didn't go into *why* I chose this project.

The reasons are fairly simple. In 2015, and in fact for decades now, software for computers has been developed almost exclusively with the aid of computers.

The sheer pervasiveness of this situation might almost lull you into an impression that it has always been done this way.

But, even if you don't know the history, you can logically reason that it can't possibly have *always* been done this way; in order to develop software using a computer, there needs to be software *on* that computer first. It's a chicken-and-egg problem.

The very first computers must have had *no* software on them, and the software for them must have been developed *without* using a

computer, and then loaded into the computer... somehow. Only after that point could the computer be used to develop more and newer software for itself.

This situation, incidentally, explains the name of this blog. Getting over this chicken-and-egg complication is called **bootstrapping**, after the mythical idea that one might reach down and, grabbing hold of one's own boot-straps, lift oneself off the ground. The concept shows up in several places in computer engineering, particularly in compiler development, but even in the common process of powering on a computer: we say the computer "boots", which is just a shortening of "bootstrap".

Meanwhile, "zero" refers to the fact that there is no software on the computer yet, and suggests an origin point; we are starting with nothing. (OK, even if that's not *absolutely* true, let's say we're trying to get as close to the "zero point of software development" as reasonably possible.)

Or to put it another way: **pen and paper is also a programming environment.**

And, since it's so much easier to develop software using a computer, it's rather a foreign environment these days. So let's revisit it, lest habituation to screens and keyboards dulls our imagination!

For the next post, I hope to write a bit about the part where the code is actually loaded into the computer (since that's the part we're going to fudge the most, so I'd like to impart what it's like, the best I can.) And while July is underway, I hope to be able to update the blog regularly with photos of the handwritten materials. Stay tuned!

Show more